



# INSIDE MACINTOSH

---

## QuickDraw GX Graphics



**Addison-Wesley Publishing Company**

Reading, Massachusetts   Menlo Park, California   New York  
Don Mills, Ontario   Wokingham, England   Amsterdam   Bonn  
Sydney   Singapore   Tokyo   Madrid   San Juan  
Paris   Seoul   Milan   Mexico City   Taipei

🍏 Apple Computer, Inc.  
© 1994 Apple Computer, Inc.  
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc. Printed in the United States of America.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple Macintosh computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for printing or clerical errors.

Apple Computer, Inc.  
20525 Mariani Avenue  
Cupertino, CA 95014  
408-996-1010

Apple, the Apple logo, LaserWriter, and Macintosh are trademarks of Apple Computer, Inc., registered in the United States and other countries.

QuickDraw and TrueType are trademarks of Apple Computer, Inc. Adobe Illustrator, Adobe Photoshop, and PostScript are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

FrameMaker is a registered trademark of Frame Technology Corporation.

Helvetica and Palatino are registered trademarks of Linotype Company.

ITC Zapf Dingbats is a registered trademark of International Typeface Corporation.

Optrotech is a trademark of Orbotech Corporation.

Simultaneously published in the United States and Canada.

#### LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

---

ISBN 0-201-40673-X  
1 2 3 4 5 6 7 8 9-CRW-9897969594  
First Printing, March 1994

#### Library of Congress Cataloging-in-Publication Data

Inside Macintosh. QuickDraw GX graphics.

p. cm.

Includes index.

ISBN 0-201-40673-X

1. Macintosh (Computer)—Programming. 2. Computer graphics.

3. QuickDraw GX. I. Apple Computer, Inc. II. Title: QuickDraw GX graphics.

QA76.8.M3153 1994

006.6'765—dc20

93-48597  
CIP

# Contents

Figures, Tables, and Listings    xi

Preface

About This Book    xxiii

---

What to Read    xxv  
Chapter Organization    xxvi  
Conventions Used in This Book    xxvi  
    Special Fonts    xxvi  
    Types of Notes    xxvii  
    Numerical Formats    xxvii  
    Type Definitions for Enumerations    xxvii  
    Illustrations    xxviii  
Development Environment    xxviii  
Developer Products and Support    xxviii

Chapter 1

Introduction to QuickDraw GX Graphics    1-1

---

About QuickDraw GX Graphics    1-4  
Geometric Shapes    1-7  
    Geometric Shape Types    1-8  
    Geometric Shape Geometries    1-9  
    Geometric Shape Fills    1-10  
    Geometric Styles, Inks, and Transforms    1-11  
    Geometric Operations    1-14  
Bitmap Shapes    1-17  
Picture Shapes    1-20

Chapter 2

Geometric Shapes    2-1

---

About Geometric Shapes    2-5  
The Geometric Properties of Shape Objects    2-7  
    Shape Type    2-7  
    Shape Geometry    2-9  
    Shape Fill    2-12  
The Geometric Shape Types    2-16  
    Empty Shapes and Full Shapes    2-16  
    Point Shapes    2-16  
    Line Shapes    2-17  
    Curve Shapes    2-18  
    Rectangle Shapes    2-20

Polygon Shapes	2-22
Path Shapes	2-25
Using Geometric Shapes	2-27
Creating and Drawing Empty Shapes and Full Shapes	2-28
Creating and Drawing Points	2-29
Creating and Drawing Lines	2-36
Creating and Drawing Curves	2-41
Creating and Drawing Rectangles	2-43
Creating and Drawing Polygons	2-45
Creating Polygons With a Single Contour	2-46
Creating Polygons With Multiple Contours	2-49
Creating Polygons With Crossed Contours	2-50
Creating and Drawing Paths	2-55
Creating Paths With a Single Contour	2-57
Creating Paths Using Only Off-Curve Points	2-59
Creating Paths With Multiple Contours	2-60
Converting Between Geometric Shape Types	2-65
Converting Shapes to Points, Lines, and Rectangles	2-66
Converting Shapes to Curve Shapes	2-71
Converting Shapes to Polygons and Paths	2-74
Replacing Geometric Points	2-79
Editing Polygon Parts	2-82
Editing Paths Parts	2-91
Editing Shape Parts	2-93
Applying Functions Described Elsewhere to Geometric Shapes	2-100
Shape-Related Functions Applicable to Geometric Shapes	2-100
Other Functions Applicable to Geometric Shapes	2-103
Geometric Shapes Reference	2-103
Data Types	2-104
The Point Structure	2-104
The Line Structure	2-105
The Curve Structure	2-105
The Rectangle Structure	2-106
Polygon Structures	2-106
Path Structures	2-107
Functions	2-108
Creating Geometric Shapes	2-109
Getting and Setting Shape Geometries	2-119
Editing Shape Geometries	2-135
Drawing Geometric Shapes	2-157
Summary of Geometric Shapes	2-163
Constants and Data Types	2-163
Functions	2-164

---

About Geometric Styles	3-5
Shapes and Styles	3-5
Incorporating Stylistic Variations Into Shape Geometries	3-8
Style Properties	3-11
Default Style Objects	3-12
Curve Error	3-14
The Geometric Pen	3-15
Style Attributes	3-17
Pen Placement	3-18
Grids	3-20
Interactions Between Caps, Joins, Dashes, and Patterns	3-22
Caps	3-23
Joins	3-25
Dashes	3-27
Patterns	3-31
Interactions Between Caps, Joins, Dashes, and Patterns	3-33
Using Geometric Styles	3-35
Associating Styles With Shapes	3-36
Constraining Shape Geometries to Grids	3-40
Constraining Shapes to Device Grids	3-42
Using Curve Error When Converting Paths to Polygons	3-45
Using Curve Error When Reducing Shapes	3-49
Manipulating Pen Width and Placement	3-51
Adding Caps to a Shape	3-57
Adding Standard Caps to a Shape	3-59
Adding Joins to a Shape	3-61
Adding Standard Joins to a Shape	3-64
Dashing a Shape	3-66
Adjusting Dashes to Fit Contours	3-70
Insetting Dashes	3-73
Breaking and Bending Dashes	3-74
Wrapping Text to a Contour	3-80
Determining Dash Positions	3-81
Adding a Pattern to a Shape	3-86
Determining Pattern Positions	3-88
Combining Caps, Joins, Dashes, and Patterns	3-91
Geometric Styles Reference	3-96
Constants and Data Types	3-96
Style Objects	3-97
Style Attributes	3-98
The Cap Structure	3-99
Cap Attributes	3-101
The Join Structure	3-101
Join Attributes	3-102

The Dash Structure	3-103
Dash Attributes	3-105
The Pattern Structure	3-106
Pattern Attributes	3-107
Functions	3-108
Getting and Setting Style Attributes	3-109
Getting and Setting Curve Error	3-114
Getting and Setting the Pen Width	3-119
Getting and Setting Caps	3-123
Getting and Setting Joins	3-129
Getting and Setting Dashes	3-134
Getting and Setting Patterns	3-142
Summary of Geometric Styles	3-149
Constants and Data Types	3-149
Functions for Manipulating Geometric Style Properties	3-151

## Chapter 4

## Geometric Operations 4-1

---

About Geometric Operations	4-4
Contours and Contour Direction	4-4
Reducing and Simplifying Shape Geometries	4-9
The Primitive Form of Shape Geometries	4-12
Geometric Information	4-16
Touching and Containing	4-18
Geometric Arithmetic	4-21
Using Geometric Operations	4-23
Determining and Reversing Contour Direction	4-23
Breaking Shape Contours	4-28
Eliminating Unnecessary Geometric Points	4-30
Simplifying Shapes	4-33
Converting a Shape to Primitive Form	4-38
Finding Geometric Information About a Shape	4-41
Finding the Length of a Contour	4-42
Finding the Point at a Certain Distance Along a Contour	4-42
Finding the Bounding Rectangle and Center Point of a Shape	4-43
Finding the Area of a Shape	4-45
Setting a Shape's Bounding Rectangle	4-47
Insetting Shapes	4-50
Determining Whether Two Shapes Touch	4-53
Determining Whether One Shape Contains Another	4-58
Performing Geometric Arithmetic With Shapes	4-60
Geometric Operations Reference	4-67
Constants and Data Types	4-67
Contour Directions	4-67

Functions	4-68
Determining and Reversing Contour Direction	4-68
Breaking Shape Contours	4-72
Reducing and Simplifying Shapes	4-74
Incorporating Style Information Into Shape Geometries	4-79
Finding Geometric Information About Shapes	4-83
Getting and Setting Shape Bounds	4-90
Insetting Shapes	4-94
Determining Whether Two Areas Touch	4-95
Determining Whether One Shape Contains Another	4-100
Performing Geometric Arithmetic With Shapes	4-104
Summary of Geometric Operations	4-117
Constants and Data Types	4-117
Functions	4-117

---

## Chapter 5      **Bitmap Shapes**      5-1

About Bitmap Shapes	5-3
Bitmap Geometries	5-5
Bitmap Styles and Inks	5-8
Bitmap Transforms	5-10
Bitmaps and View Devices	5-12
Using Bitmap Shapes	5-14
Creating and Drawing Bitmaps	5-15
Creating Black-and-White Bitmaps	5-15
Creating Color Bitmaps	5-21
Dithering and Halftoning Bitmaps	5-30
Applying Transfer Modes to Bitmaps	5-32
Converting Other Types of Shapes to Bitmaps	5-34
Applying Transformations to Bitmaps	5-38
Mapping Bitmap Shapes	5-39
Clipping Bitmap Shapes	5-43
Creating Bitmaps With Disk-Based Pixel Images	5-44
Creating Bitmaps Offscreen	5-45
Editing Part of a Bitmap	5-53
Applying Functions Described Elsewhere to Bitmap Shapes	5-54
Functions That Post Errors or Warnings When Applied to Bitmap Shapes	5-55
Shape-Related Functions Applicable to Bitmap Shapes	5-56
Geometric Operations Applicable to Bitmap Shapes	5-58
Style-Related Functions Applicable to Bitmap Shapes	5-59
Ink-Related Functions Applicable to Bitmap Shapes	5-59
Transform-Related Functions Applicable to Bitmap Shapes	5-59
View-Related Functions Applicable to Bitmap Shapes	5-61

Bitmap Shapes Reference	5-61
Constants and Data Types	5-61
The Bitmap Geometry Structure	5-62
The Long Rectangle Structure	5-64
Constants For Bitmaps With Disk-Based Pixel Images	5-64
Bitmap Data Source Alias Structure	5-65
Functions	5-65
Creating Bitmaps	5-65
Getting and Setting Bitmap Geometries	5-68
Editing Bitmaps	5-71
Drawing Bitmaps	5-76
Checking Bitmap Colors	5-79
Summary of Bitmap Shapes	5-81
Constants and Data Types	5-81
Functions	5-82

## Chapter 6

## Picture Shapes 6-1

---

About Picture Shapes	6-3
Overriding Styles, Inks, and Transforms	6-8
Multiple References	6-10
Unique Items Shape Attribute	6-15
Picture Hierarchies	6-18
Transform Concatenation	6-19
About Hit-Testing Picture Shapes	6-24
Using Picture Shapes	6-26
Creating and Drawing Picture Shapes	6-27
Getting and Setting Picture Geometries	6-31
Adding Items to a Picture	6-32
Removing and Replacing Items in a Picture	6-35
Using Overriding Styles, Inks, and Transforms	6-38
Adding Multiple References	6-40
Adding Items With the Unique Items Attribute Set	6-43
Creating Picture Hierarchies	6-44
Hit-Testing Pictures	6-46
Applying Functions Described Elsewhere to Picture Shapes	6-52
Functions That Post Errors or Warnings When Applied to Pictures	6-52
Shape-Related Functions Applicable to Pictures	6-54
Geometric Operations Applicable to Pictures	6-55
Style-Related Functions Applicable to Pictures	6-55
Ink-Related Functions Applicable to Pictures	6-56
Transform-Related Functions Applicable to Pictures	6-56
Picture Shapes Reference	6-57
Functions	6-57
Creating Picture Shapes	6-57
Getting and Setting Picture Geometries	6-59



Editing Picture Parts	6-63
Drawing Pictures	6-67
Hit-Testing Pictures	6-69
Summary of Picture Shapes	6-72
Functions	6-72

Glossary	GL-1
----------	------

---

Index	IN-1
-------	------

---



# Figures, Tables, and Listings

## Color Plates

---

*Color plates are immediately preceding the title page.*

<b>Color Plate 1</b>	The effect of transfer modes on bitmap shapes
<b>Color Plate 2</b>	A blended color ramp bitmap
<b>Color Plate 3</b>	A bitmap with an eight-color color set
<b>Color Plate 4</b>	A color ramp bitmap
<b>Color Plate 5</b>	Transformed bitmaps
<b>Color Plate 6</b>	A bitmap drawn with and without a transfer mode

## Preface

## About This Book      xxiii

---

<b>Figure P-1</b>	Roadmap to the QuickDraw GX suite of books	xxiv
-------------------	--	------

## Chapter 1

## Introduction to QuickDraw GX Graphics      1-1

---

<b>Figure 1-1</b>	Shape object structure	1-5
<b>Figure 1-2</b>	The geometric shape types and examples of geometric shape geometries	1-8
<b>Figure 1-3</b>	A polygon shape with a single polygon contour containing three geometric points	1-10
<b>Figure 1-4</b>	Framed shapes versus solid shapes	1-11
<b>Figure 1-5</b>	Two condensed views of a polygon shape	1-12
<b>Figure 1-6</b>	The geometric style properties and some examples of their effects	1-13
<b>Figure 1-7</b>	An example of reducing a shape	1-14
<b>Figure 1-8</b>	An example of simplifying a shape	1-14
<b>Figure 1-9</b>	Some examples of the geometric information available about a shape	1-15
<b>Figure 1-10</b>	Some examples of the geometric arithmetic you can perform with shapes	1-16
<b>Figure 1-11</b>	Sample bitmap shapes	1-17
<b>Figure 1-12</b>	A bitmap shape	1-18
<b>Figure 1-13</b>	Elements of a bitmap geometry	1-19
<b>Figure 1-14</b>	Sample picture shapes	1-20
<b>Figure 1-15</b>	A picture hierarchy	1-21
<b>Table 1-1</b>	Where to find information on shape-type conversion	1-6

## Chapter 2

## Geometric Shapes      2-1

---

<b>Figure 2-1</b>	A shape object	2-6
<b>Figure 2-2</b>	The geometric shape types and examples of geometric shape geometries	2-8

<b>Figure 2-3</b>	A polygon shape with a single contour containing three geometric points	2-10
<b>Figure 2-4</b>	Framed shapes versus solid shapes	2-12
<b>Figure 2-5</b>	The various shape fills and examples of their effects	2-13
<b>Figure 2-6</b>	The even-odd rule and winding-number rule algorithms	2-14
<b>Figure 2-7</b>	The inverse even-odd shape fill	2-15
<b>Figure 2-8</b>	Two lines	2-17
<b>Figure 2-9</b>	A quadratic Bézier curve	2-18
<b>Figure 2-10</b>	Finding the midpoint of a curve	2-19
<b>Figure 2-11</b>	Dividing a curve into two smaller curves	2-20
<b>Figure 2-12</b>	A rectangle geometry shown framed and filled	2-21
<b>Figure 2-13</b>	A polygon shape with two polygon contours	2-23
<b>Figure 2-14</b>	A polygon drawn with the even-odd and winding shape fills	2-24
<b>Figure 2-15</b>	A path with two consecutive off-curve points	2-25
<b>Figure 2-16</b>	A path shape filled with the even-odd and winding shape fills	2-26
<b>Figure 2-17</b>	A point	2-30
<b>Figure 2-18</b>	Two different point geometries	2-35
<b>Figure 2-19</b>	A line	2-37
<b>Figure 2-20</b>	Parallel lines	2-39
<b>Figure 2-21</b>	Nearly parallel lines	2-40
<b>Figure 2-22</b>	A curve	2-42
<b>Figure 2-23</b>	A rectangle	2-44
<b>Figure 2-24</b>	A framed rectangle	2-45
<b>Figure 2-25</b>	A polygon	2-47
<b>Figure 2-26</b>	A triangular polygon with inverse shape fill	2-47
<b>Figure 2-27</b>	A filled polygon with two separate contours	2-50
<b>Figure 2-28</b>	A framed polygon with a crossed contour	2-51
<b>Figure 2-29</b>	A solid polygon with a crossed contour	2-51
<b>Figure 2-30</b>	A polygon with an overlapping contour and closed-frame shape fill	2-53
<b>Figure 2-31</b>	A polygon with an overlapping contour and even-odd shape fill	2-53
<b>Figure 2-32</b>	A polygon with an overlapping contour and winding shape fill	2-54
<b>Figure 2-33</b>	A path	2-58
<b>Figure 2-34</b>	A round path shape	2-60
<b>Figure 2-35</b>	A path shape with two concentric clockwise contours and closed-frame shape fill	2-62
<b>Figure 2-36</b>	A path shape with two concentric clockwise contours and even-odd shape fill	2-63
<b>Figure 2-37</b>	A path shape with two concentric clockwise contours and winding shape fill	2-63
<b>Figure 2-38</b>	A path shape with an internal counterclockwise contour and closed-frame shape fill	2-64
<b>Figure 2-39</b>	A path shape with even-odd or winding shape fill	2-65
<b>Figure 2-40</b>	A figure-eight path shape	2-67
<b>Figure 2-41</b>	A path shape before and after conversion to a rectangle shape	2-68
<b>Figure 2-42</b>	A path shape before and after conversion to a line shape	2-69
<b>Figure 2-43</b>	A path shape before and after conversion to a point shape	2-70
<b>Figure 2-44</b>	A line shape before and after conversion to a curve shape	2-72

<b>Figure 2-45</b>	A rectangle shape before and after conversion to a curve shape 2-73
<b>Figure 2-46</b>	A polygon shape before and after conversion to a curve shape 2-74
<b>Figure 2-47</b>	A rectangle shape before and after conversion to a polygon shape 2-75
<b>Figure 2-48</b>	A path shape before and after conversion to a polygon shape 2-77
<b>Figure 2-49</b>	Polygon shape with two contours before and after conversion to a path shape 2-79
<b>Figure 2-50</b>	A path shape with a flat top 2-81
<b>Figure 2-51</b>	A path shape with geometric points replaced 2-81
<b>Figure 2-52</b>	A polygon shape with two contours 2-83
<b>Figure 2-53</b>	A polygon shape extracted from a larger polygon shape 2-85
<b>Figure 2-54</b>	A polygon with two geometric points replaced by a single geometric point 2-87
<b>Figure 2-55</b>	A polygon shape with one contour 2-87
<b>Figure 2-56</b>	A polygon shape edited with the <code>gxBreakNeitherEdit</code> flag set 2-89
<b>Figure 2-57</b>	A polygon shape edited with the <code>gxBreakLeftEdit</code> flag set 2-89
<b>Figure 2-58</b>	A polygon shape edited with the <code>gxBreakRightEdit</code> flag set 2-90
<b>Figure 2-59</b>	A path shape with two curved contours 2-92
<b>Figure 2-60</b>	A path shape edited with <code>GXSetPathParts</code> 2-93
<b>Figure 2-61</b>	A path shape with a flat top 2-95
<b>Figure 2-62</b>	A path shape edited to have a pointy top 2-96
<b>Figure 2-63</b>	A path shape edited to have a round top 2-97
<b>Figure 2-64</b>	A diagonal line 2-99
<b>Figure 2-65</b>	An edited line 2-99
 <b>Table 2-1</b>	 Shape-related functions that exhibit special behavior with geometric shapes 2-101
 <b>Listing 2-1</b>	 Drawing a point without creating a point shape 2-30
<b>Listing 2-2</b>	Creating a point shape with the <code>GXNewPoint</code> function 2-31
<b>Listing 2-3</b>	Creating a point shape with the <code>GXNewShapeVector</code> function 2-32
<b>Listing 2-4</b>	Creating a point shape with the <code>GXNewShape</code> and <code>GXSetPoint</code> functions 2-33
<b>Listing 2-5</b>	Using the <code>GXSetPoint</code> function to replace a point shape's geometry 2-34
<b>Listing 2-6</b>	Drawing a line without creating a line shape 2-37
<b>Listing 2-7</b>	Creating a line shape with the <code>GXNewLine</code> function 2-38
<b>Listing 2-8</b>	Drawing two parallel lines 2-39
<b>Listing 2-9</b>	Creating a curve shape 2-41
<b>Listing 2-10</b>	Creating a rectangle shape 2-43
<b>Listing 2-11</b>	Creating a framed rectangle 2-44
<b>Listing 2-12</b>	Drawing a triangular polygon 2-46
<b>Listing 2-13</b>	Creating a polygon with two contours 2-49
<b>Listing 2-14</b>	Creating a polygon with a crossed contour 2-50
<b>Listing 2-15</b>	Creating a polygon with an overlapping contour 2-52
<b>Listing 2-16</b>	Drawing a path shape 2-57

<b>Listing 2-17</b>	Creating a path using only off-curve control points	2-59
<b>Listing 2-18</b>	Creating a path with concentric contours	2-61
<b>Listing 2-19</b>	Creating a figure-eight path shape	2-67
<b>Listing 2-20</b>	Converting a line to a curve	2-71
<b>Listing 2-21</b>	Converting a rectangle to a curve	2-72
<b>Listing 2-22</b>	Converting a polygon shape to a curve shape	2-73
<b>Listing 2-23</b>	Converting a rectangle shape to a polygon shape	2-75
<b>Listing 2-24</b>	Converting a path shape to a polygon shape	2-76
<b>Listing 2-25</b>	Converting a polygon shape to a path shape	2-78
<b>Listing 2-26</b>	Replacing geometric points	2-80
<b>Listing 2-27</b>	Creating a polygon shape with two contours	2-82
<b>Listing 2-28</b>	Extracting part of a polygon shape	2-84
<b>Listing 2-29</b>	Replacing geometric points of a polygon shape	2-86
<b>Listing 2-30</b>	Inserting a geometric point in a polygon shape	2-88
<b>Listing 2-31</b>	Creating a path shape with two curved contours	2-91
<b>Listing 2-32</b>	Creating a path shape with one contour	2-94
<b>Listing 2-33</b>	Creating a diagonal line	2-98

## Chapter 3

## Geometric Styles 3-1

---

<b>Figure 3-1</b>	Style object with geometric properties highlighted	3-6
<b>Figure 3-2</b>	Shared style objects	3-7
<b>Figure 3-3</b>	Effects of the <code>GXPrimitiveShape</code> function on a line shape	3-9
<b>Figure 3-4</b>	Effects of the <code>GXPrimitiveShape</code> function on a rectangle shape	3-10
<b>Figure 3-5</b>	The QuickDraw GX geometric pen	3-15
<b>Figure 3-6</b>	Differing pen widths	3-16
<b>Figure 3-7</b>	Pixels included in a hairline	3-16
<b>Figure 3-8</b>	A geometry with no hairline	3-17
<b>Figure 3-9</b>	Pen placement	3-18
<b>Figure 3-10</b>	Effect of the auto-inset style attribute	3-19
<b>Figure 3-11</b>	Effect of the auto-inset and inside-frame style attributes for a crossed contour	3-19
<b>Figure 3-12</b>	Eliminating crossed contours	3-20
<b>Figure 3-13</b>	Constraining shapes to grids	3-21
<b>Figure 3-14</b>	Caps, joins, dashes, and patterns	3-22
<b>Figure 3-15</b>	A shape with caps	3-23
<b>Figure 3-16</b>	A shape with level caps	3-24
<b>Figure 3-17</b>	Standard cap shapes	3-24
<b>Figure 3-18</b>	A shape with joins	3-25
<b>Figure 3-19</b>	A shape with level joins	3-26
<b>Figure 3-20</b>	Standard joins	3-26
<b>Figure 3-21</b>	Sharp join with miter	3-27
<b>Figure 3-22</b>	A dashed shape	3-27
<b>Figure 3-23</b>	Scaling a dash shape	3-28
<b>Figure 3-24</b>	Effect of the clip dash attribute	3-29
<b>Figure 3-25</b>	Effects of breaking a dash	3-30
<b>Figure 3-26</b>	Effects of bending a dash	3-30
<b>Figure 3-27</b>	A shape with a pattern	3-31
<b>Figure 3-28</b>	Pattern placed on a nonrectilinear grid	3-32

<b>Figure 3-29</b>	Effects of the port-align pattern attribute	3-32
<b>Figure 3-30</b>	Effects of the port-map pattern attribute	3-33
<b>Figure 3-31</b>	A shape with a cap, join, and pattern	3-34
<b>Figure 3-32</b>	A shape with a dash and a pattern	3-34
<b>Figure 3-33</b>	A shape with a clipped dash and a cap and join	3-35
<b>Figure 3-34</b>	Rectangle with thick pen	3-38
<b>Figure 3-35</b>	Scaled, but not constrained, V shape	3-41
<b>Figure 3-36</b>	Constrained V shape	3-42
<b>Figure 3-37</b>	Rotated star not constrained to device grid (magnified 200 percent)	3-44
<b>Figure 3-38</b>	Rotated star constrained to device grid (magnified 200 percent)	3-45
<b>Figure 3-39</b>	Polygon approximation of a circle with curve error of 1	3-46
<b>Figure 3-40</b>	Polygon approximation of a circle with curve error of 5	3-47
<b>Figure 3-41</b>	Polygon approximation of a circle with curve error of 10	3-47
<b>Figure 3-42</b>	Polygon resulting from a curve error of 0	3-48
<b>Figure 3-43</b>	Wavy line	3-50
<b>Figure 3-44</b>	Wavy line somewhat smoothed by curve error of 10	3-50
<b>Figure 3-45</b>	Wavy line smoothed by curve error of 15	3-50
<b>Figure 3-46</b>	Wavy line completely straightened by curve error of 20	3-50
<b>Figure 3-47</b>	A hairline figure eight	3-52
<b>Figure 3-48</b>	A thick figure eight	3-52
<b>Figure 3-49</b>	A figure eight with pen inset	3-53
<b>Figure 3-50</b>	A figure eight with pen outset	3-54
<b>Figure 3-51</b>	A reversed figure eight with pen outset	3-55
<b>Figure 3-52</b>	Uncrossed figure eight with pen outset	3-56
<b>Figure 3-53</b>	An arrow	3-59
<b>Figure 3-54</b>	Round and square caps	3-61
<b>Figure 3-55</b>	A square with diamond-shaped joins	3-63
<b>Figure 3-56</b>	A square with level joins	3-63
<b>Figure 3-57</b>	An angle with a sharp join	3-65
<b>Figure 3-58</b>	An angle with a truncated sharp join	3-65
<b>Figure 3-59</b>	A dashed curve	3-68
<b>Figure 3-60</b>	A curve with scaled dashes	3-68
<b>Figure 3-61</b>	A curve with clipped dashes	3-69
<b>Figure 3-62</b>	A curve with phased dashes	3-69
<b>Figure 3-63</b>	Circle dashed with diamonds	3-71
<b>Figure 3-64</b>	Circle with automatically advanced dashes	3-72
<b>Figure 3-65</b>	Circle with diamond dashes inset	3-73
<b>Figure 3-66</b>	Circle with diamond dashes moved toward the center	3-74
<b>Figure 3-67</b>	Dash shape with two contours	3-75
<b>Figure 3-68</b>	Circle dashed with double diamonds	3-76
<b>Figure 3-69</b>	Circle with dashes broken	3-77
<b>Figure 3-70</b>	Circle with hairline dashes	3-78
<b>Figure 3-71</b>	Circle with bent hairline dashes	3-79
<b>Figure 3-72</b>	Wrapped text	3-81
<b>Figure 3-73</b>	Dash positions for a clock	3-83
<b>Figure 3-74</b>	A clock shape	3-85
<b>Figure 3-75</b>	A rectangle with a pattern	3-87
<b>Figure 3-76</b>	A framed rectangle with a pattern	3-88
<b>Figure 3-77</b>	Shape with changing pattern	3-91

<b>Figure 3-78</b>	Angle shape with cap, join, and pattern	3-93
<b>Figure 3-79</b>	Angle shape with dash and pattern; caps and join ignored	3-94
<b>Figure 3-80</b>	Shape with cap, join, dash, and the clip dash attribute set	3-95
<b>Listing 3-1</b>	Adding style information by directly manipulating a style object	3-37
<b>Listing 3-2</b>	Manipulating style information indirectly	3-39
<b>Listing 3-3</b>	Constraining a shape to a half-inch grid	3-40
<b>Listing 3-4</b>	Creating a shape with fractional geometric point positions	3-43
<b>Listing 3-5</b>	Converting a circle to a polygon	3-46
<b>Listing 3-6</b>	Creating a complicated contour	3-49
<b>Listing 3-7</b>	Defining a figure eight	3-51
<b>Listing 3-8</b>	Removing unwanted contour crossings	3-55
<b>Listing 3-9</b>	Creating an arrow	3-57
<b>Listing 3-10</b>	Adding round caps and square caps to a curve	3-60
<b>Listing 3-11</b>	Adding joins to a shape	3-61
<b>Listing 3-12</b>	Adding a sharp join to an angle shape	3-64
<b>Listing 3-13</b>	Creating a curve shape dashed with diamonds	3-66
<b>Listing 3-14</b>	Creating a dashed circle	3-70
<b>Listing 3-15</b>	Creating a dash with multiple contours	3-75
<b>Listing 3-16</b>	Wrapping text	3-80
<b>Listing 3-17</b>	Creating a circle with 12 dashes	3-82
<b>Listing 3-18</b>	Creating a clock shape	3-83
<b>Listing 3-19</b>	Patterning a shape	3-86
<b>Listing 3-20</b>	Changing a pattern throughout a patterned shape	3-89
<b>Listing 3-21</b>	Combining a cap, join, and pattern	3-92

## Chapter 4

## Geometric Operations 4-1

---

<b>Figure 4-1</b>	Line contours	4-5
<b>Figure 4-2</b>	A path shape with two contours	4-6
<b>Figure 4-3</b>	A path whose contour direction is not immediately obvious	4-7
<b>Figure 4-4</b>	A path whose inner contour has the same contour direction as its outer contour	4-8
<b>Figure 4-5</b>	A path shape whose inner and outer contours have different contour directions	4-8
<b>Figure 4-6</b>	Effects of reducing and simplifying shape geometries	4-10
<b>Figure 4-7</b>	How simplifying a shape can produce more predictable results when drawing	4-11
<b>Figure 4-8</b>	Simple example of the <code>GXPrimitiveShape</code> function	4-13
<b>Figure 4-9</b>	More involved example of the <code>GXPrimitiveShape</code> function	4-15
<b>Figure 4-10</b>	Geometric information available about a path shape	4-17
<b>Figure 4-11</b>	A path shape resized by changing its bounding rectangle	4-18
<b>Figure 4-12</b>	Testing whether one shape touches another	4-19
<b>Figure 4-13</b>	Testing whether one shape contains another	4-20
<b>Figure 4-14</b>	Geometric arithmetic with two solid shapes	4-21
<b>Figure 4-15</b>	Geometric arithmetic with a framed shape and a solid shape	4-22
<b>Figure 4-16</b>	Geometric inversion	4-22



<b>Figure 4-17</b>	A polygon shape whose two contours have opposite contour directions 4-25
<b>Figure 4-18</b>	A polygon shape with the direction of both contours reversed 4-26
<b>Figure 4-19</b>	A polygon shape with the direction of the inner contour reversed 4-27
<b>Figure 4-20</b>	A path shape with a single contour 4-29
<b>Figure 4-21</b>	A path shape broken into two contours 4-29
<b>Figure 4-22</b>	A polygon shape with unnecessary geometric points 4-31
<b>Figure 4-23</b>	A polygon shape with the unnecessary geometric points removed 4-32
<b>Figure 4-24</b>	A polygon shape with a crossed contour 4-34
<b>Figure 4-25</b>	A polygon shape with no crossed contours 4-34
<b>Figure 4-26</b>	A path shape with two concentric clockwise contours and even-odd shape fill 4-36
<b>Figure 4-27</b>	A path shape with two concentric contours with opposite contour direction 4-36
<b>Figure 4-28</b>	A path shape with two concentric clockwise contours and winding shape fill 4-37
<b>Figure 4-29</b>	A path shape simplified to a single clockwise contour 4-37
<b>Figure 4-30</b>	A hourglass-shaped polygon with a thick border 4-39
<b>Figure 4-31</b>	A polygon shape with style information incorporated into its geometry 4-39
<b>Figure 4-32</b>	The primitive form of the polygon shape after simplification 4-40
<b>Figure 4-33</b>	A path with an outer clockwise contour and an inner counterclockwise contour 4-42
<b>Figure 4-34</b>	Finding a specified point on a path contour 4-43
<b>Figure 4-35</b>	Finding the bounding rectangle and the center point of a path 4-44
<b>Figure 4-36</b>	Finding the center point of two contours 4-44
<b>Figure 4-37</b>	Finding the area of a path, two contours with same contour direction 4-45
<b>Figure 4-38</b>	Finding the area of a path, two contours with opposite contour direction 4-46
<b>Figure 4-39</b>	Finding the area of a simplified path 4-46
<b>Figure 4-40</b>	A circular path 4-48
<b>Figure 4-41</b>	A circular path after bounding rectangle changed 4-48
<b>Figure 4-42</b>	A path shape with a transform mapping 4-49
<b>Figure 4-43</b>	A tight curve 4-51
<b>Figure 4-44</b>	An inset curve shape 4-51
<b>Figure 4-45</b>	An outset curve 4-52
<b>Figure 4-46</b>	A rectangle containing a circular path 4-54
<b>Figure 4-47</b>	A rectangle that touches a circular path shape 4-55
<b>Figure 4-48</b>	A rectangle and a circular path touching at a single point 4-56
<b>Figure 4-49</b>	A large circular path shape touching a smaller circular path shape 4-57
<b>Figure 4-50</b>	A path shape with two contours and a smaller concentric rectangle shape 4-59
<b>Figure 4-51</b>	A diamond-shaped polygon geometry and a circular path geometry 4-61
<b>Figure 4-52</b>	The intersection of a diamond-shaped polygon and a circular path 4-61

<b>Figure 4-53</b>	The union of a diamond-shaped polygon and a circular path 4-62
<b>Figure 4-54</b>	The union of a framed diamond-shaped polygon and a circular path 4-63
<b>Figure 4-55</b>	The result of subtracting a circular path from a diamond-shaped polygon 4-63
<b>Figure 4-56</b>	The result of subtracting a diamond-shaped polygon from a circular path 4-64
<b>Figure 4-57</b>	The result of the exclusive-OR operation on a polygon and a path 4-65
<b>Figure 4-58</b>	An inverted diamond 4-66
<b>Listing 4-1</b>	Creating a polygon shape with two contours having opposite contour directions 4-24
<b>Listing 4-2</b>	Creating a path shape with a single contour 4-28
<b>Listing 4-3</b>	Creating a polygon with redundant geometric points 4-31
<b>Listing 4-4</b>	Creating a polygon shape with a crossed contour 4-33
<b>Listing 4-5</b>	Creating a path shape with two clockwise contours 4-35
<b>Listing 4-6</b>	Creating an hourglass polygon shape with a thick pen width 4-38
<b>Listing 4-7</b>	Creating a path shape with two contours having opposite contour directions 4-41
<b>Listing 4-8</b>	Creating a circular path 4-47
<b>Listing 4-9</b>	Creating a tight curve shape 4-50
<b>Listing 4-10</b>	Creating a rectangle and a circular path shape 4-53
<b>Listing 4-11</b>	Creating a path shape with two contours and a smaller concentric rectangle shape 4-58
<b>Listing 4-12</b>	Creating a diamond-shaped polygon and a circular path that intersect 4-60

## Chapter 5

## Bitmap Shapes 5-1

---

<b>Figure 5-1</b>	A bitmap shape 5-4
<b>Figure 5-2</b>	A black-and-white bitmap geometry 5-6
<b>Figure 5-3</b>	A grayscale bitmap geometry 5-7
<b>Figure 5-4</b>	The effect of transfer modes on bitmap shapes 5-9
<b>Figure 5-5</b>	The effect of mappings on bitmap shapes 5-10
<b>Figure 5-6</b>	The effect of the <code>gxMapTransformShape</code> shape attribute on bitmap mappings 5-11
<b>Figure 5-7</b>	Bitmaps and view devices 5-13
<b>Figure 5-8</b>	A black-and-white bitmap—32 bits wide 5-17
<b>Figure 5-9</b>	An example of unaligned bytes per row 5-19
<b>Figure 5-10</b>	An envelope with a shadow 5-20
<b>Figure 5-11</b>	A bitmap with a grayscale color set (four shades) 5-22
<b>Figure 5-12</b>	A bitmap with a grayscale color set (sixteen shades) 5-23
<b>Figure 5-13</b>	A bitmap with an eight-color color set 5-24
<b>Figure 5-14</b>	A color ramp from red to green 5-28
<b>Figure 5-15</b>	Dithered bitmaps 5-31
<b>Figure 5-16</b>	Halftoned bitmaps 5-32
<b>Figure 5-17</b>	A blended color ramp 5-34
<b>Figure 5-18</b>	A bitmap representation of a path shape 5-36
<b>Figure 5-19</b>	A bitmap and its bounding rectangle 5-36

<b>Figure 5-20</b>	A bitmap drawn over a background	5-37
<b>Figure 5-21</b>	A bitmap with a transfer mode drawn over a background	5-38
<b>Figure 5-22</b>	A path shape converted to a bitmap shape	5-39
<b>Figure 5-23</b>	A path shape converted to a bitmap shape and then skewed	5-39
<b>Figure 5-24</b>	A color ramp bitmap	5-40
<b>Figure 5-25</b>	A bitmap after multiple transformations	5-40
<b>Figure 5-26</b>	Scaled text	5-41
<b>Figure 5-27</b>	Scaled text and a scaled bitmap	5-42
<b>Figure 5-28</b>	A clipped bitmap	5-43
<b>Figure 5-29</b>	Multiple shapes drawn to a bitmap	5-51
<b>Figure 5-30</b>	An extracted bitmap	5-53
<b>Figure 5-31</b>	An edited bitmap	5-54
<b>Table 5-1</b>	Shape-editing functions that post errors or warnings when applied to bitmaps	5-55
<b>Table 5-2</b>	Geometric operations that post errors or warnings when applied to bitmaps	5-56
<b>Table 5-3</b>	Shape-related functions that exhibit special behavior when applied to bitmaps	5-57
<b>Table 5-4</b>	Geometric operations that exhibit special behavior when applied to bitmaps	5-58
<b>Table 5-5</b>	Transform-related functions that exhibit special behavior when applied to bitmaps	5-60
<b>Table 5-6</b>	View-related functions that can be applied to bitmaps	5-61
<b>Listing 5-1</b>	Creating a black-and-white bitmap	5-15
<b>Listing 5-2</b>	A bit image with an even number of bytes per row	5-20
<b>Listing 5-3</b>	Defining a color set	5-23
<b>Listing 5-4</b>	Creating a color ramp	5-26
<b>Listing 5-5</b>	Creating a color ramp using the ramp library	5-28
<b>Listing 5-6</b>	Creating a color ramp using both the ramp and color libraries	5-29
<b>Listing 5-7</b>	Halftoning a bitmap	5-31
<b>Listing 5-8</b>	Applying a transfer mode to a bitmap	5-33
<b>Listing 5-9</b>	Converting a path to a bitmap	5-35
<b>Listing 5-10</b>	Scaling text	5-41
<b>Listing 5-11</b>	Scaling a bitmap	5-42
<b>Listing 5-12</b>	Creating a black-and-white bitmap	5-46
<b>Listing 5-13</b>	Creating an offscreen bitmap	5-49
<b>Listing 5-14</b>	Creating an offscreen bitmap using the offscreen library	5-51

## Chapter 6

### Picture Shapes 6-1

---

<b>Figure 6-1</b>	A picture shape	6-4
<b>Figure 6-2</b>	A picture item	6-5
<b>Figure 6-3</b>	A picture geometry with two items	6-6
<b>Figure 6-4</b>	Condensed view of picture with two items	6-7
<b>Figure 6-5</b>	A picture shape with overrides	6-9
<b>Figure 6-6</b>	A picture containing multiple references to the same shape	6-10

<b>Figure 6-7</b>	A condensed view of a picture with multiple references	6-11
<b>Figure 6-8</b>	Multiple references with overriding transforms	6-12
<b>Figure 6-9</b>	Multiple references with overriding styles, inks, and transforms	6-14
<b>Figure 6-10</b>	An empty picture shape and a polygon shape	6-15
<b>Figure 6-11</b>	Adding a polygon shape to a picture shape	6-16
<b>Figure 6-12</b>	Adding a shape to a picture twice	6-17
<b>Figure 6-13</b>	A condensed view of a picture hierarchy	6-18
<b>Figure 6-14</b>	A path shape and its transform	6-19
<b>Figure 6-15</b>	A picture with an overriding transform	6-20
<b>Figure 6-16</b>	Simple transform concatenation	6-21
<b>Figure 6-17</b>	Intricate transform concatenation	6-23
<b>Figure 6-18</b>	A picture shape and hit-test points	6-25
<b>Figure 6-19</b>	A picture of a house with a roof and a door	6-29
<b>Figure 6-20</b>	A picture of a house with a relocated door	6-32
<b>Figure 6-21</b>	A house with a lawn, walkway, and chimney	6-35
<b>Figure 6-22</b>	A house with chimney removed	6-36
<b>Figure 6-23</b>	A house with the chimney replaced	6-37
<b>Figure 6-24</b>	A house picture with an overriding style, ink, and transform	6-40
<b>Figure 6-25</b>	A house with four windows	6-42
<b>Figure 6-26</b>	A house with four windows and four unique overriding transforms	6-44
<b>Figure 6-27</b>	A house rotated by 90 degrees two times	6-45
<b>Figure 6-28</b>	Grounds picture	6-47
<b>Figure 6-29</b>	House picture	6-47
<b>Figure 6-30</b>	Picture containing grounds picture and house picture	6-48
<b>Figure 6-31</b>	Hit-testing the picture of house and grounds	6-49
<b>Figure 6-32</b>	Hit-testing the picture at depth 2 and level 1	6-50
<b>Table 6-1</b>	Hit-testing a picture at different depths and levels	6-51
<b>Table 6-2</b>	Geometric operations that post errors or warnings when applied to pictures	6-53
<b>Table 6-3</b>	Shape-related functions that exhibit special behavior when applied to pictures	6-54
<b>Table 6-4</b>	Geometric operations that exhibit special behavior when applied to pictures	6-55
<b>Listing 6-1</b>	Creating a simple picture of a house	6-28
<b>Listing 6-2</b>	Disposing of shapes contained in a picture before disposing of the picture	6-30
<b>Listing 6-3</b>	Extracting and editing items from a picture	6-31
<b>Listing 6-4</b>	Defining new shapes for the house picture	6-33
<b>Listing 6-5</b>	Adding new shapes to the house picture	6-34
<b>Listing 6-6</b>	Removing an item from a picture	6-36
<b>Listing 6-7</b>	Replacing one shape with another	6-37
<b>Listing 6-8</b>	Creating style, ink, and transform objects	6-38
<b>Listing 6-9</b>	Creating a picture whose items have overriding styles, inks, and transforms	6-39
<b>Listing 6-10</b>	Disposing of overriding style, ink, and transform objects before drawing	6-40
<b>Listing 6-11</b>	Adding four items that reference the same shape to a house picture	6-41

<b>Listing 6-12</b>	Disposing of the white rectangle and the three transform objects before drawing	6-42
<b>Listing 6-13</b>	Adding unique items to a picture	6-43
<b>Listing 6-14</b>	Creating a picture hierarchy	6-45
<b>Listing 6-15</b>	Creating a picture hierarchy	6-46
<b>Listing 6-16</b>	Hit-testing a picture shape	6-49



## About This Book

---

QuickDraw GX is an integrated, object-based approach to graphics programming on Macintosh computers. This book, *Inside Macintosh: QuickDraw GX Graphics*, describes the data types and functions you use to create graphic images.

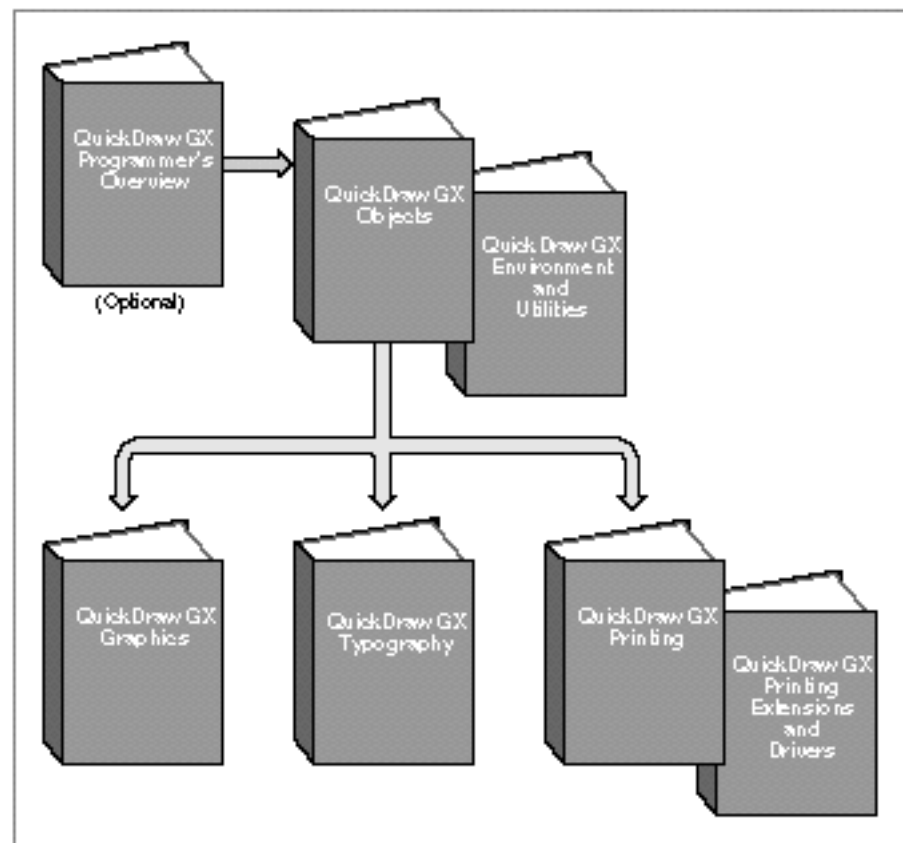
For application programming purposes, QuickDraw GX augments the capabilities of some of the Macintosh system software managers documented in other parts of *Inside Macintosh*. In situations where your application uses QuickDraw GX for drawing, information in this book replaces much of the information in *Inside Macintosh: Imaging With QuickDraw*. QuickDraw and QuickDraw GX coexist without conflict, however, and you can use both in the same program. Furthermore, for tasks outside the scope of QuickDraw GX, such as managing graphics ports, you need to use QuickDraw.

Before you read this book, you should already be familiar with information described elsewhere in the *Inside Macintosh* QuickDraw GX suite of books. In particular, you should be familiar with much of the information in *Inside Macintosh: QuickDraw GX Objects*. You should read the information about QuickDraw GX shapes and objects in the chapter “Introduction to QuickDraw GX” in that book. You should also read the chapters “Shape Objects,” “Style Objects,” “Ink Objects,” and “Transform Objects” in that book.

## P R E F A C E

For an alternative approach to learning QuickDraw GX, you can read *QuickDraw GX Programmer's Overview* before or along with this book. *QuickDraw GX Programmer's Overview* teaches QuickDraw GX programming through building extensive code samples. Figure P-1 shows the suggested reading order for the QuickDraw GX books.

**Figure P-1** Roadmap to the QuickDraw GX suite of books





## What to Read

---

This book describes three types of QuickDraw GX shapes you can use to make graphic images:

- n geometric shapes
- n bitmap shapes
- n picture shapes

The other types of QuickDraw GX shapes (the typographic shapes) are discussed in *Inside Macintosh: QuickDraw GX Typography*.

The chapters of this book cover these topics:

- n Geometric shapes, which are the building blocks for graphics. These shapes, which include points, lines, curves, rectangles, polygons, and paths, make up the graphic elements supported by most drawing programs. The chapter “Geometric Shapes” in this book describes geometric shapes in detail.
- n Geometric styles, which are the stylistic variations you can make to geometric shapes. The chapter “Geometric Styles” in this book describes these variations.
- n Geometric operations, which are the functions you can use to manipulate geometric shapes and obtain geometric information about geometric shapes. The chapter “Geometric Operations” in this book describes these functions.
- n Bitmap shapes, which contain pixel images. These shapes allow you to create graphics by specifying the color value of each pixel in the image. The chapter “Bitmap Shapes” in this book describes bitmap shapes in detail. This chapter also references a number of the color plates you can find at the front of this book.
- n Picture shapes, which are collections of QuickDraw GX shapes, including other picture shapes. You can find this type of shape described in the chapter “Picture Shapes,” in this book.

## Chapter Organization

---

Most chapters in this book follow a standard general structure. For example, the chapter “Geometric Shapes” contains these major sections:

- n “About Geometric Shapes.” This section provides an overview of geometric shapes.
- n “Using Geometric Shapes.” This section describes how you can create and manipulate geometric shapes using QuickDraw GX. It describes how to use the most common functions, gives related user interface information, provides code samples, and supplies additional information.
- n “Geometric Shapes Reference.” This section provides a complete reference to geometric shapes by describing the constants, data types, and functions that you use with geometric shapes. Each function description follows a standard format, which gives the function declaration; a description of every parameter; the function result, if any; and a list of errors, warnings, and notices. Most function descriptions give additional information about using the function and include cross-references to related information elsewhere.
- n “Summary of Geometric Shapes.” This shows the C interface for the constants, data types, and functions associated with geometric shapes.

## Conventions Used in This Book

---

This book uses various conventions to present certain types of information.

### Special Fonts

---

All code listings, reserved words, and the names of data structures, constants, fields, parameters, and functions are shown in `Courier` (`this is Courier`).

When new terms are introduced, they are in **boldface**. These terms are also defined in the glossary.

## Types of Notes

---

There are several types of notes used in this book.

### **Note**

A note formatted like this contains information that is interesting but possibly not essential to an understanding of the main text. The wording in the title may say something more descriptive than just “Note,” for example “Implementation Note.” (An example appears on page 2-22.) u

### **IMPORTANT**

A note like this contains information that is especially important. (An example appears on page 2-28.) s

## Numerical Formats

---

Hexadecimal numbers are shown in this format: 0x0008.

The numerical values of constants are shown in decimal, unless the constants are flag or mask elements that can be summed, in which case they are shown in hexadecimal.

## Type Definitions for Enumerations

---

Enumeration declarations in this book are commonly followed by a type definition that is not strictly part of the enumeration. You can use the type to specify one of the enumerated values for a parameter or field. The type name is usually the singular of the enumeration name, as in the following example:

```
enum gxDashAttributes {
    gxBendDash          = 0x0001,
    gxBreakDash         = 0x0002,
    gxClipDash          = 0x0004,
    gxLevelDash         = 0x0008,
    gxAutoAdvanceDash = 0x0010
};
typedef long gxDashAttribute;
```

## Illustrations

---

The following conventions are used in illustrations in this book.

In illustrations that show object properties, properties that are object references are in *italics*.

In order to focus attention on the key part of some drawings, other parts are printed in gray, rather than black.

This book also uses other conventions for representing shape objects, style objects, ink objects, and transform objects.

See Figure 1-1, Figure 1-2, and Figure 1-6 in Chapter 1, “Introduction to QuickDraw GX Graphics,” for examples of these conventions.

## Development Environment

---

The QuickDraw GX functions described in this book are available using C interfaces. How you access these functions depends on the development environment you are using.

Code listings in this book are shown in ANSI C. They suggest methods of using various functions and illustrate techniques for accomplishing particular tasks. Although most code listings have been compiled and tested, Apple Computer, Inc., does not intend for you to use these code samples in your applications.

## Developer Products and Support

---

APDA is Apple’s worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most of our products. APDA offers convenient payment and shipping options, including site licensing.

## P R E F A C E

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA  
Apple Computer, Inc.  
P.O. Box 319  
Buffalo, NY 14207-0319

Telephone	800-282-2732 (United States) 800-637-0029 (Canada) 716-871-6555 (International)
-----------	---

Fax	716-871-6511
-----	--------------

AppleLink	APDA
-----------	------

America Online	APDAorder
----------------	-----------

CompuServe	76666,2405
------------	------------

Internet	APDA@applelink.apple.com
----------	--------------------------

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.



# Introduction to QuickDraw GX Graphics

---

## Contents

About QuickDraw GX Graphics	1-4
Geometric Shapes	1-7
Geometric Shape Types	1-8
Geometric Shape Geometries	1-9
Geometric Shape Fills	1-10
Geometric Styles, Inks, and Transforms	1-11
Geometric Operations	1-14
Bitmap Shapes	1-17
Picture Shapes	1-20





## Introduction to QuickDraw GX Graphics

This chapter introduces the main concepts found in the rest of this book and gives an overview of the three types of QuickDraw GX shapes you can use to make graphic images:

- n geometric shapes
- n bitmap shapes
- n picture shapes

The other types of QuickDraw GX shapes (the typographic shapes) are discussed in *Inside Macintosh: QuickDraw GX Typography*.

You should be familiar with information described elsewhere in the *Inside Macintosh: QuickDraw GX* books before you read this chapter. In particular, you should read the information about QuickDraw GX shapes and objects in the chapter “Introduction to QuickDraw GX” in *Inside Macintosh: QuickDraw GX Objects*. You should also read the chapter “Shape Objects” in that book.

As you read this chapter and the other chapters in this book, you might want to be familiar with the other information in *Inside Macintosh: QuickDraw GX Objects*—in particular, you might also read the “Style Objects,” “Ink Objects,” and “Transform Objects” chapters in that book.

The next section reviews the objects that make up a QuickDraw GX shape and introduces the different types of graphic shapes. The remaining sections of this chapter briefly discuss

- n the structure of geometric shapes
- n the contents of geometric shape geometries
- n the shape fill property and how it affects geometric shapes
- n the properties of the style object that modify geometric shapes
- n the geometric operations provided by QuickDraw GX
- n the structure of bitmap shapes
- n the structure of picture shapes

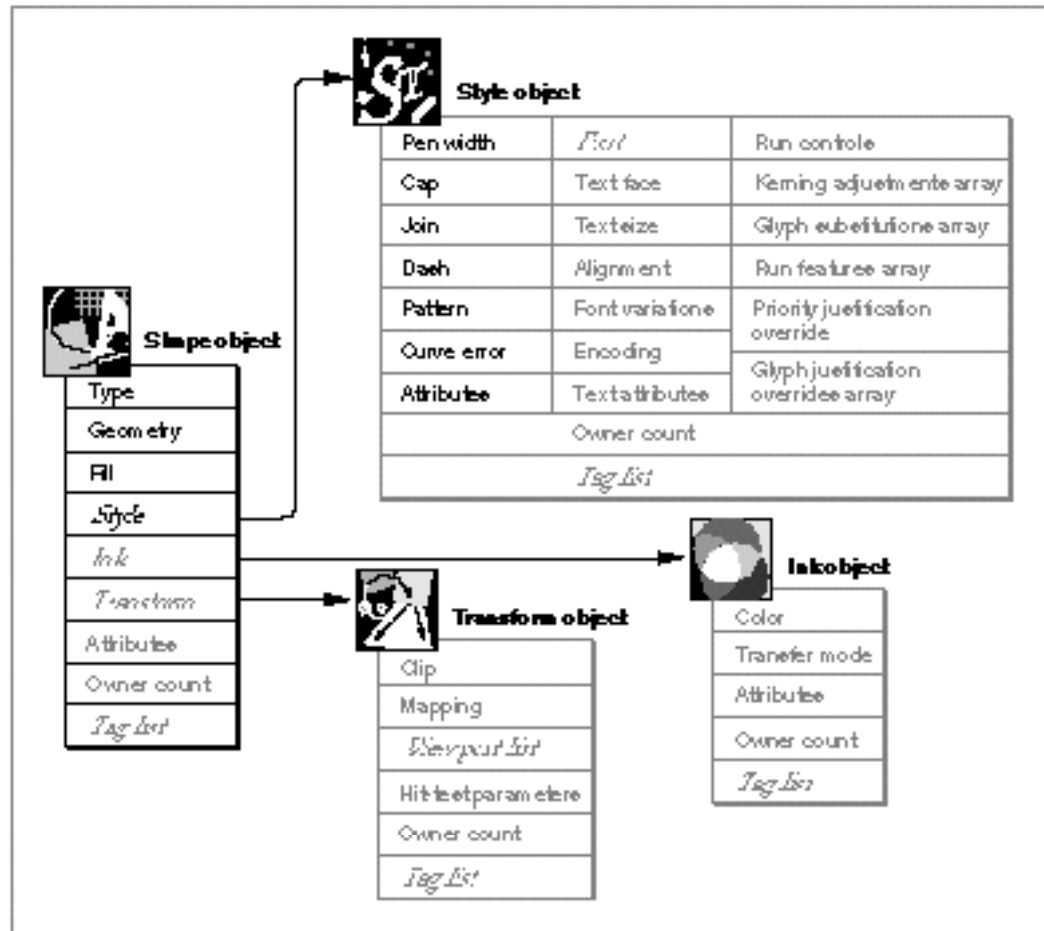
## About QuickDraw GX Graphics

---

With QuickDraw GX, you create graphics by creating QuickDraw GX shapes. Graphics shapes include geometric shapes, bitmap shapes, and picture shapes:

- n **Geometric shapes** are the building blocks for graphics. These shapes, which include points, lines, curves, rectangles, polygons, and paths, make up the graphic elements supported by most drawing programs. There are also two special types of geometric shapes: empty shapes, which cover no area, and full shapes, which cover all area.
- n **Bitmap shapes** contain pixel images. These shapes allow you to create graphics by specifying the color value of each pixel in the image.
- n **Picture shapes** are collections of QuickDraw GX shapes, including other picture shapes.

All QuickDraw GX shapes share the same basic structure. They are all represented by a shape object and its associated style, ink, and transform objects. Figure 1-1 shows the four basic QuickDraw GX objects and lists the properties of each. This figure includes all of the properties of these objects. However, this book examines only a subset of these properties. Properties not examined in this book are grayed out.

**Figure 1-1** Shape object structure

Like all shapes, geometric shapes are represented by a shape object in memory. Three of the properties of the shape object—shape type, shape geometry, and shape fill—and how they apply to geometric shapes in particular, are introduced in the section “Geometric Shapes” beginning on page 1-7 and are fully discussed in the chapter “Geometric Shapes” in this book.

Geometric shapes use the style object properties highlighted in Figure 1-1. These properties are introduced in the section “Geometric Styles, Inks, and Transforms” beginning on page 1-11 and are fully examined in the chapter “Geometric Styles” in this book.

Geometric shapes also use the properties of their ink and transform objects. You can find more information about these objects in the chapters “Ink Objects” and “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

Bitmap shapes use their shape, style, ink, and transform objects, although they make limited use of some of the properties of these objects. Bitmap shapes are introduced in the section “Bitmap Shapes” beginning on page 1-17 and are fully examined in the chapter “Bitmap Shapes” in this book.

Picture shapes use their shape and transform objects, but do not use their style or ink objects. Picture shapes are introduced in the section “Picture Shapes” beginning on page 1-20 and are fully examined in the chapter “Picture Shapes” in this book.

QuickDraw GX allows you to convert between the different types of shapes. Table 1-1 describes where to look in each book for information regarding each possible kind of conversion.

**Table 1-1** Where to find information on shape-type conversion

	To a geometric shape	To a bitmap shape	To a picture shape	To a typographic shape
<b>From a geometric shape</b>	See “Geometric Shapes” in this book	See “Bitmap Shapes” in this book	See “Picture Shapes” in this book	(not possible)
<b>From a bitmap shape</b>	(not possible)	See “Bitmap Shapes” in this book	See “Picture Shapes” in this book	(not possible)
<b>From a picture shape</b>	(not possible)	See “Bitmap Shapes” in this book	See “Picture Shapes” in this book	(not possible)
<b>From a typographic shape</b>	See “Typographic Shapes” in <i>QuickDraw GX Typography</i>	See “Bitmap Shapes” in this book	See “Picture Shapes” in this book	See “Typographic Shapes” in <i>QuickDraw GX Typography</i>

## Geometric Shapes

---

QuickDraw GX provides eight types of geometric shapes—the basic building blocks of QuickDraw GX graphics. These shapes include empty shapes, full shapes, points, lines, rectangles, curves, polygons, and paths. You can use these shapes for drawing, for calculating areas, for clipping, as elements of more complex graphics, and so on.

As with all types of QuickDraw GX shapes, a geometric shape is represented by a shape object in QuickDraw GX memory. However, what defines a geometric shape—what makes it different from other types of shapes—is how it uses the properties of the shape object:

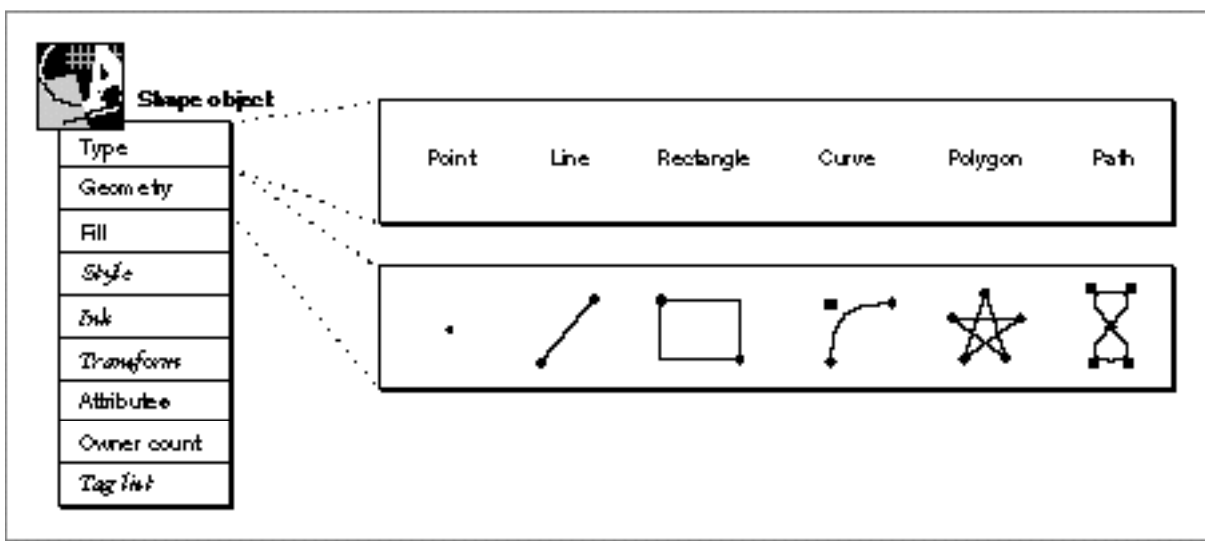
- n The shape type property specifies the type of the geometric shape—empty, full, point, line, curve, rectangle, polygon, or path.
- n The geometry property specifies the positions of the points that define the shape—for example, the end points of a line, or the corners of a rectangle.
- n The shape fill property specifies how the geometry of the shape is interpreted—for example, as a framed outline or as a solid area.
- n The style property references a style object, which specifies modifications to the geometric shape—for example, pen width, dashes, and patterns.
- n The ink and transform properties reference an ink and a transform object. The ink object specifies the color and transfer mode applied to the shape when drawn. The transform object specifies mapping transformations made to the shape, how the shape is clipped, how the shape is hit-tested, and to what view ports the shape is finally drawn.
- n The attributes, owner count, and tag list properties contain object-related information about the shape. These properties affect how the shape object is maintained in memory, when the memory held by the shape is freed, and other information you might want to specify for a particular shape.

Geometric shapes use all of the shape properties—to understand geometric shapes fully, you should be familiar with all of these properties, which are introduced in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*. The way that geometric shapes use these properties differently from other types of shapes is described in this book, particularly in the chapters “Geometric Shapes” and “Geometric Styles.”

## Geometric Shape Types

There are six basic types of geometric shapes and two special types. The basic geometric shapes include points, lines, rectangles, curves, polygons, and paths; the two special types are empty shapes and full shapes. Figure 1-2 lists the basic geometric shape types and also shows a sample geometry for each of them. Each geometry is made up of geometric points and edges that connect the geometric points. The next section, “Geometric Shape Geometries,” introduces these concepts in more detail.

**Figure 1-2** The geometric shape types and examples of geometric shape geometries



The empty shape and the full shape are not shown in this figure. An empty shape is a shape that has no geometry and covers no area. A full shape is the inverse of an empty shape—it covers all area. For a complete description of each type of geometric shape, see the chapter “Geometric Shapes” in this book.

## Geometric Shape Geometries

---

Each type of geometric shape uses the geometry property of its shape object in a slightly different manner. For example, empty shapes and full shapes store no information in their geometry, because they require no further geometry information—their shape type says it all.

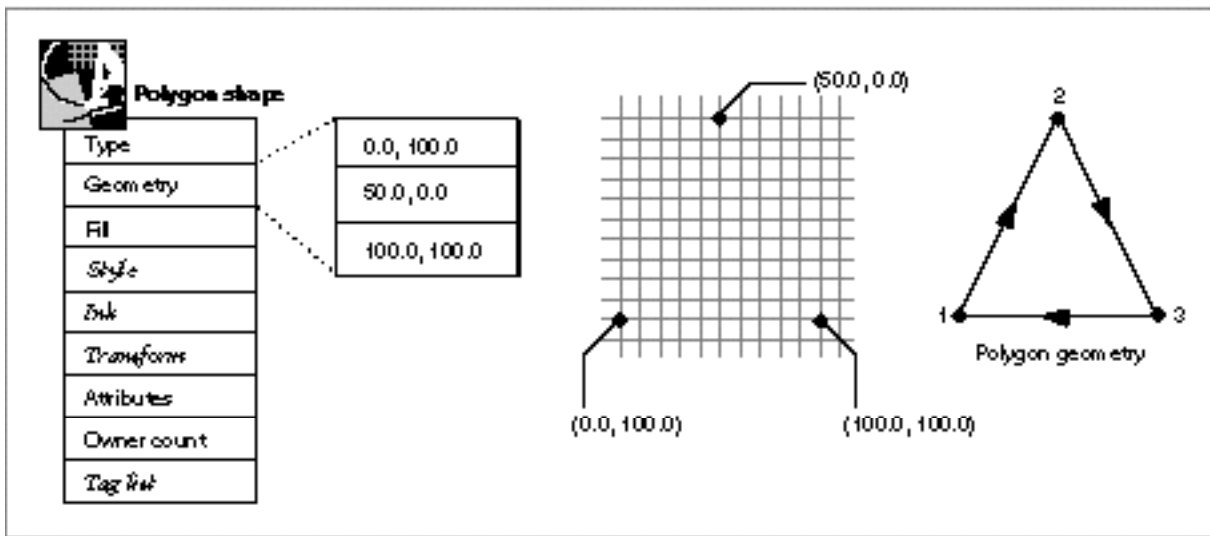
However, for other types of geometric shapes, the shape type does not contain all the geometry information necessary to define the shape. The geometries of these shapes contain (x, y) coordinate pairs called **geometric points**—points that specify the location, dimension, and form of the geometric shapes:

- n Point geometries contain one geometric point—an x-coordinate and a y-coordinate—to specify the position of the point shape.
- n Line geometries contain two geometric points—one point to specify where the line starts and one to specify where the line ends.
- n Rectangle geometries also contain two geometric points—one point to specify one corner of the rectangle, and another point to specify the opposing corner.
- n Curve shapes store three geometric points in their geometry—one to specify where the curve starts, another to specify where the curve ends, and another, called the **off-curve control point**, to specify the tangents used to define the curve.
- n Polygon geometries are made up of zero, one, or more polygon contours. Each **polygon contour** is series of geometric points connected by straight edges.
- n Path geometries are similiar to polygon geometries, but path geometries also store information about which geometric points are on-curve and which are off-curve control points. Therefore, **path contours** can have curves as well as straight lines.

For more information about the geometries of each geometric shape type, see the chapter “Goemetric Shapes” in this book.

Figure 1-3 shows a polygon shape with a single polygon contour made up of three geometric points. This figure shows three views of the polygon geometry: as a list of (x, y) coordinate pairs, as three geometric points plotted on a geometric grid, and as three points connected by three straight lines. This third way of viewing geometries is used frequently throughout this book, as it shows not only the geometric points, but also the implied **edges** that connect them. Notice that geometric points have fixed-point coordinates—you can specify fractional positions.

**Figure 1-3** A polygon shape with a single polygon contour containing three geometric points



## Geometric Shape Fills

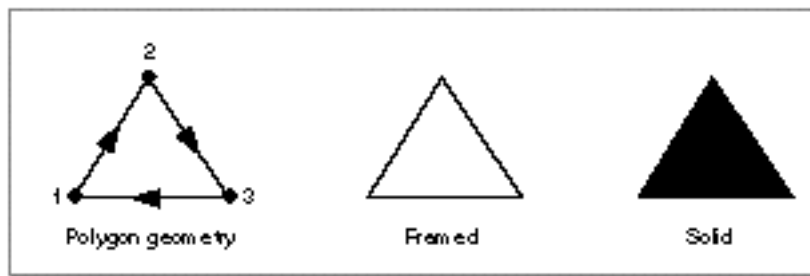
The **shape fill** property specifies how QuickDraw GX interprets the geometric points of a geometric shape's geometry. There are two basic types of shape fills:

- n **Framed fills.** These shape fills indicate that QuickDraw GX should interpret the shape as an outline—as a series of edges.
- n **Solid fills.** These shape fills indicate that QuickDraw GX should interpret the shape as a solid area—the edges of the shape represent the boundaries of the area.



Figure 1-4 shows an example of a polygon contour similar to the one in Figure 1-3, and how QuickDraw GX might draw it with a framed fill and with a solid fill.

**Figure 1-4** Framed shapes versus solid shapes



For more information about the various kinds of shape fills provided by QuickDraw GX, see the chapter “Geometric Shapes” in this book.

## Geometric Styles, Inks, and Transforms

Like all QuickDraw GX shapes, geometric shapes reference a style object, an ink object, and a transform object. Figure 1-5 shows a condensed view of how a polygon shape might use these four objects.

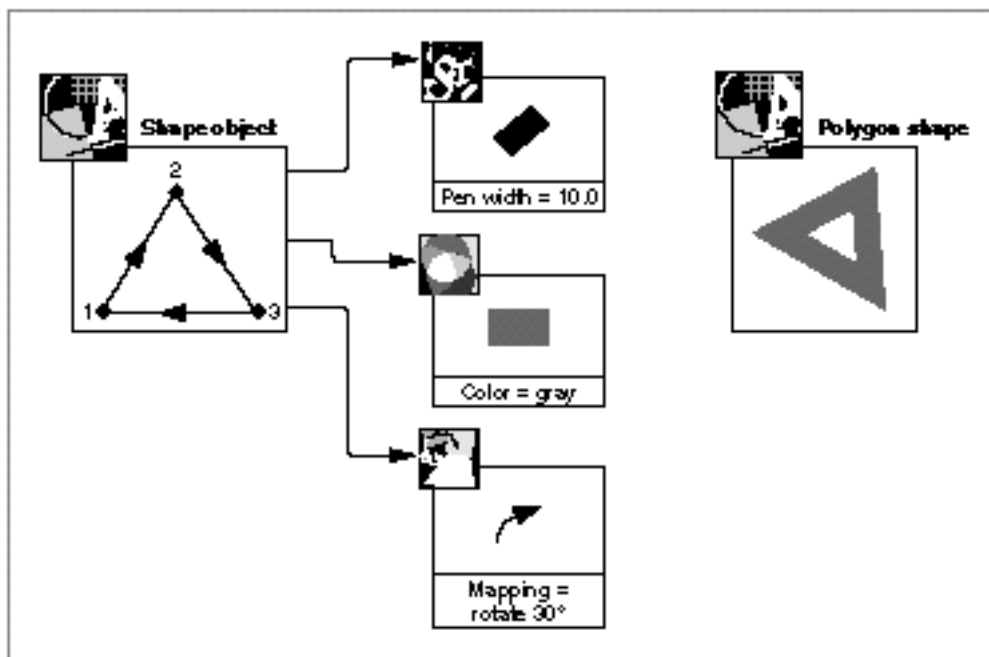
Instead of listing every property of each of these objects, the first half of Figure 1-5 (the left side) depicts a single important property for each object:

- n For the shape object, it shows the polygon geometry.
- n For the style object, it shows the pen width.
- n For the ink object, it shows the color.
- n For the transform object, it shows the transformation mapping.

This condensed view of these objects is used frequently throughout this book to highlight information important to a particular example.

The second half of Figure 1-5 (the right side) shows an even more condensed view of the polygon shape. In this view, all of the stylistic, color, and transform variations have been incorporated into the shape itself—basically showing the shape as it is drawn. This extremely condensed view is used occasionally throughout this book, particularly when many shapes must appear in a single figure, as in the chapter “Picture Shapes.”

**Figure 1-5** Two condensed views of a polygon shape



Because the ink and transform objects are used in the same way by geometric and typographic shapes, these two objects are discussed in *Inside Macintosh: QuickDraw GX Objects*, rather than in this book.

However, geometric shapes use their style objects in a very different way than typographic shapes do.

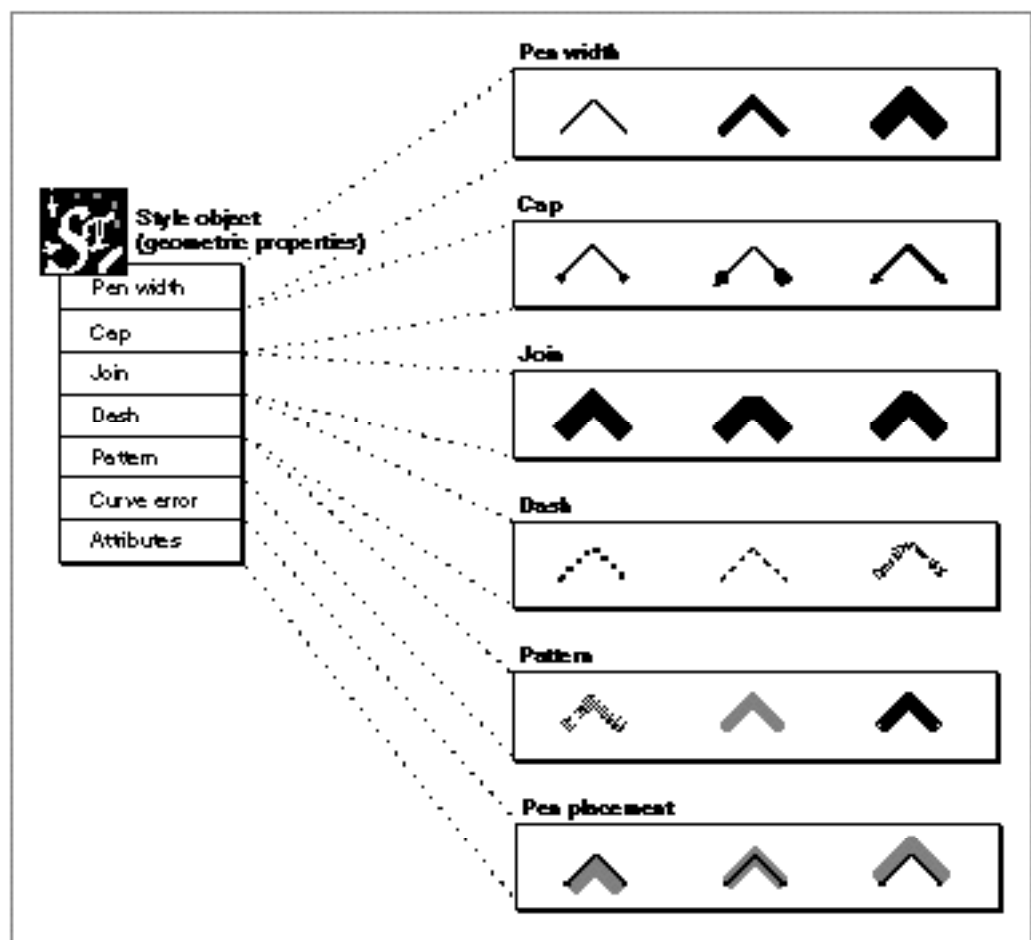
The style object has three types of properties:

- n **Object-related style properties**, which are discussed in the chapter “Style Objects” in *Inside Macintosh: QuickDraw GX Objects*. These properties apply to the style as an object in memory.
- n **Typographic style properties**, which are discussed in the chapter “Typographic Styles” in *Inside Macintosh: QuickDraw GX Typography*. These properties apply only to typographic shapes.
- n **Geometric style properties**, which are discussed in the chapter “Geometric Styles” in this book. These properties apply primarily to geometric shapes.

The geometric style properties are the properties of the style object that specify modifications to geometric shapes. With these properties, you can specify how wide QuickDraw GX should draw a shape's edges, whether the edges should be solid or dashed, whether corners should be round or sharp, what pattern should fill a shape's area, and so on.

Figure 1-6 shows the geometric properties of the style object. This figure also gives examples of the effects of these properties.

**Figure 1-6** The geometric style properties and some examples of their effects



## Geometric Operations

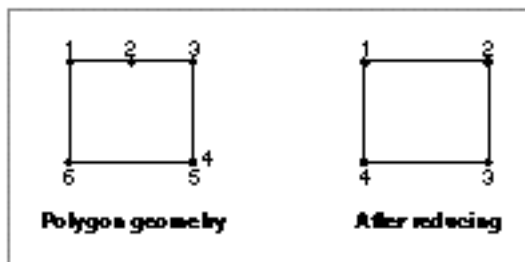
QuickDraw GX provides functions that allow you to modify the geometries of geometric shapes, obtain information about their geometries, and combine the geometries of two shapes.

One such geometric operation allows you to remove unnecessary or redundant geometric points from the shape's geometry—this process is called **reducing** a geometry.

Figure 1-7 shows a polygon geometry with two unnecessary geometric points:

- n Point 2 lies on the same line as points 1 and 3, and therefore has no effect on the geometry.
- n Points 4 and 5 lie on top of one another, and so only one of them is necessary for this geometry.

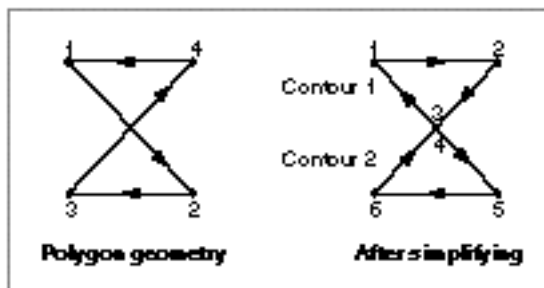
**Figure 1-7** An example of reducing a shape



In addition to unnecessary geometric points, a shape geometry can have a number of other complicating qualities, such as crossed edges or overlapping contours.

QuickDraw GX provides a geometric operation that redefines a shape's geometry to eliminate these qualities. This process is called **simplifying** a shape. Figure 1-8 shows a polygon contour with two edges that cross and the result of simplifying this shape.

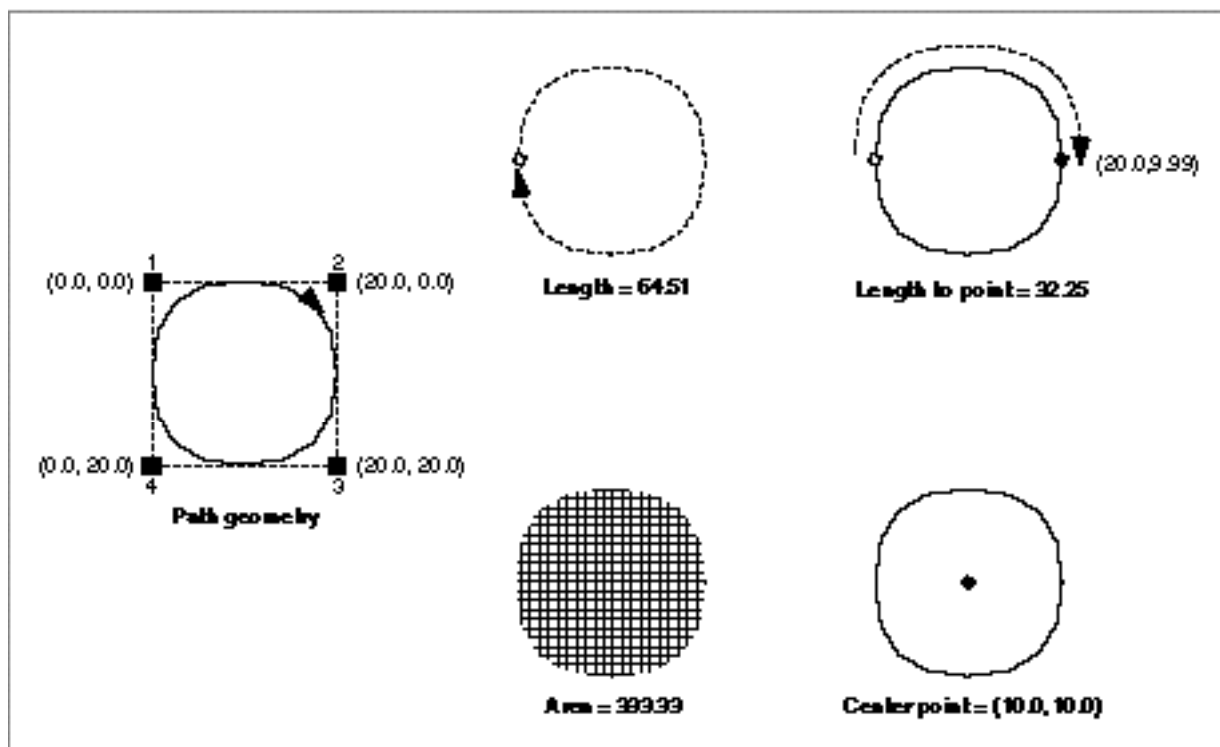
**Figure 1-8** An example of simplifying a shape



As Figure 1-8 shows, simplifying the polygon geometry splits it into two contours: an upper triangular contour with three geometric points, and a lower triangular contour with three geometric points. Although the simplified geometry contains more geometric points and more contours than the original, it does not contain any crossed edges.

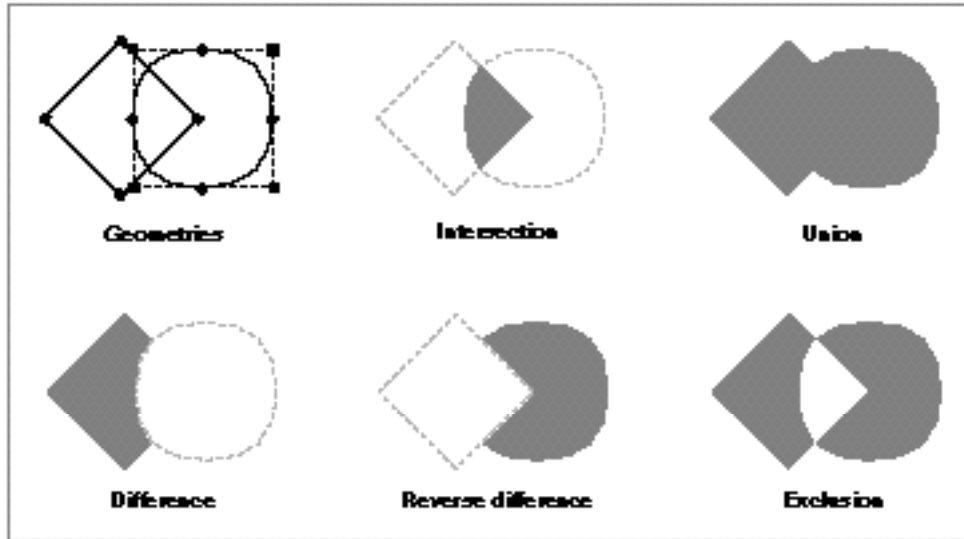
You can find more about reducing and simplifying shape geometries in the chapter “Geometric Operations” in this book. That chapter also describes many functions that allow you to obtain information about geometric shapes and perform geometric arithmetic on them. Figure 1-9 shows some examples of the different types of geometric information that QuickDraw GX calculates for you.

**Figure 1-9** Some examples of the geometric information available about a shape



You can find more about geometric information in the “Geometric Operations” chapter of this book.

Another important type of geometric operation is geometric arithmetic. Figure 1-10 shows examples of intersection, union, difference, reverse difference, and exclusion operations, which each return a result calculated by combining the geometries of two shapes in different ways.

**Figure 1-10** Some examples of the geometric arithmetic you can perform with shapes

Other geometric operations provided by QuickDraw GX allow you to

- n alter the order of the geometric points specified in a shape's geometry
- n break a single shape contour into multiple contours
- n calculate whether two shapes intersect
- n calculate whether one shape contains another shape
- n inset the geometric points of a shape's geometry
- n scale the shape to fit in a new bounding rectangle
- n invert the geometry of a shape

These geometric operations are all discussed in the chapter "Geometric Operations" in this book.

The chapter "Transform Objects" in *Inside Macintosh: QuickDraw GX Objects* describes a related set of functions you can use to perform geometric modifications to a shape's geometry. These functions allow you to

- n move a shape
- n rotate a shape
- n scale a shape
- n skew a shape
- n perform any arbitrary mapping on a shape

Depending on the setting of a shape's map-transform shape attribute, these functions either modify the mapping matrix contained in the shape's transform object or recalculate the geometric points contained in the shape's geometry directly.

## Bitmap Shapes

---

**Bitmap shapes** allow you to create images for which you specify the color value of each pixel. Geometric shapes create images with more flexibility—they can be rendered by QuickDraw GX accurately at any output device resolution. However, you might still want to use bitmap shapes for a number of reasons. For example, if you know the resolution of an output device, you can create a bitmap shape to use as an offscreen graphics buffer. As another example, since bitmaps allow you to specify multiple colors within a single shape, you can use bitmaps to create gradients, or ramps—shapes that fade from one color to another.

Figure 1-11 shows some sample bitmaps.

---

**Figure 1-11** Sample bitmap shapes



Although there are many types of geometric shapes—points, lines, curves, and so on—there is only one type of bitmap shape. Bitmap shapes make extensive use of their geometry property. In fact, most of the information useful to bitmap shapes is stored in their geometry—the values of the bitmap’s pixels, the dimensions of the bitmap, and the color information used by the bitmap.

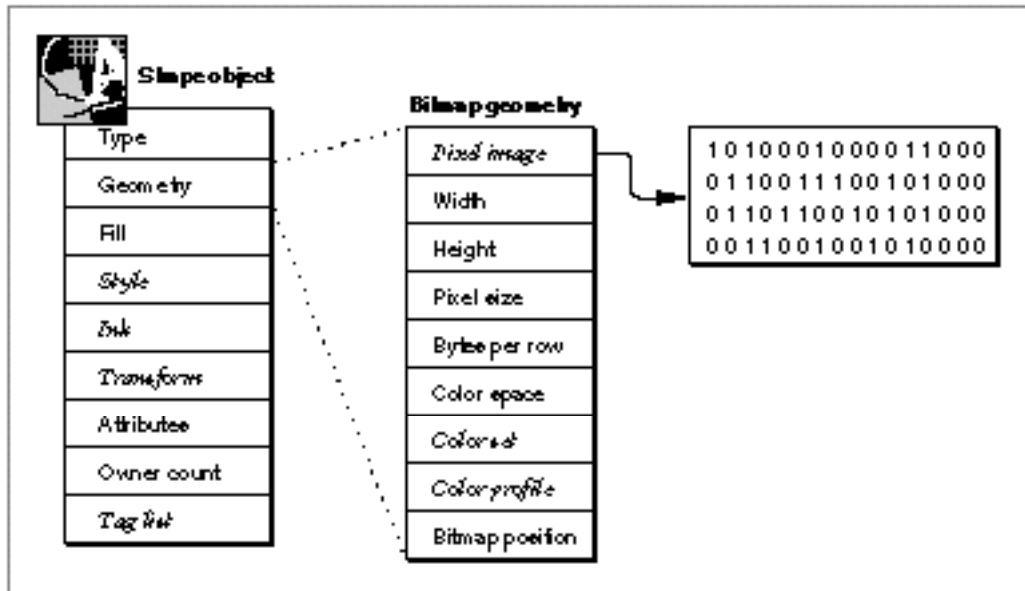
Bitmap shapes don’t make much use of their shape fill property, and they use very little of their associated style object. In fact, the only pieces of information in a style object used by bitmap shapes are the style attributes that determine whether the upper-left corner of the bitmap should be constrained to an integer grid position.

Because bitmap shapes store their own color information in their geometries, they don’t use the color property of their ink object. They do, however, use the transfer mode property of their ink objects.

Bitmap shapes make full use of their transform objects. For example, you can scale, skew, rotate, and clip bitmap shapes. You can also hit-test bitmap shapes, but you cannot hit-test parts of a bitmap shape as you can for other types of shapes. For more information about transform objects and hit-testing, see the chapter “Transform Objects” and the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects*.

Figure 1-12 shows a bitmap shape object and bitmap geometry.

**Figure 1-12** A bitmap shape



As Figure 1-12 shows, a bitmap geometry contains a reference to a **pixel image**, which contains the color values of each pixel in the bitmap. QuickDraw GX allows pixel images to be stored in three locations:

- n in memory allocated by your application
- n in memory allocated and managed by QuickDraw GX
- n in a disk file

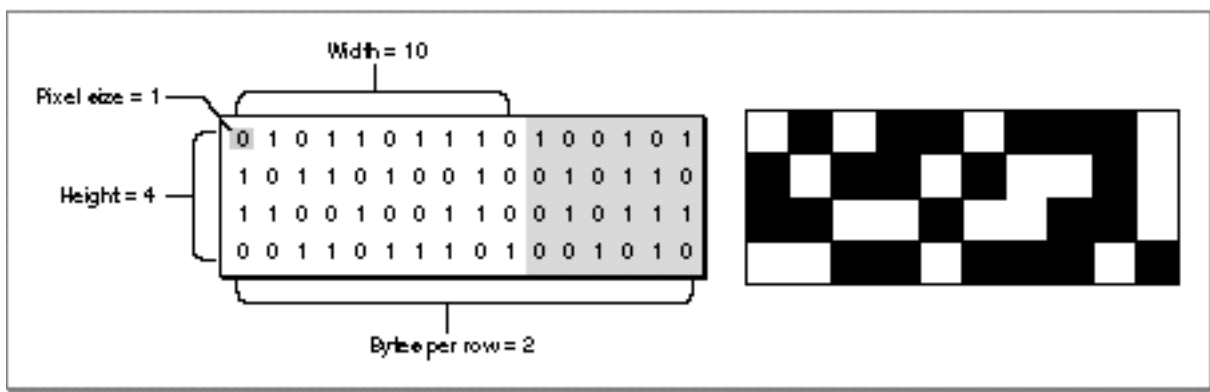
Each of these options presents different advantages and disadvantages. For example, storing a pixel image in a disk file allows you to have large bitmaps without keeping the entire pixel image in memory. However, QuickDraw GX provides only limited access to this type of pixel image: it can read the image, but cannot make changes to it.



Different bitmap shapes may reference the same pixel image. You might want to use this feature to draw the same pixel image with two different transfer modes, for example, or to draw the same pixel image in two different color spaces.

The other fields of a bitmap geometry define the dimensions, color information, and position of the bitmap's pixel image. Figure 1-13 shows a sample bitmap geometry that uses one bit to represent each pixel, and has four rows and ten columns. Since each row of the pixel image requires only ten bits, the pixel image is padded so that each row is represented by an even number of bytes.

**Figure 1-13** Elements of a bitmap geometry



The color space and color set fields of the bitmap geometry allow you to specify how QuickDraw GX should interpret the pixel values. In this example, pixel values of 0 represent white pixels and pixel values of 1 represent black pixels.

The color profile field specifies color-matching information. See the chapter “Color-Related Objects” in *Inside Macintosh: QuickDraw GX Objects* for more information about color values, color spaces, color sets, and color matching.

For more information about bitmap shapes, see the chapter “Bitmap Shapes” in this book.

## Picture Shapes

---

**Picture shapes** contain collections of other shapes. They allow you to gather disparate elements together inside a single shape.

You can use picture shapes for many reasons, including to group a page of shapes together for printing, to provide a grouping feature in a graphics application, or to simplify your programming by gathering a number of shapes together and applying modifications to the group as a whole.

Figure 1-14 shows three sample picture shapes:

- n The first picture shape combines a number of geometric shapes—rectangles, polygons, and paths—into one picture.
- n The second picture shape includes a bitmap shape as well—the lawn is a gradient, or ramp, which fades from dark to light.
- n The third picture shape includes typographic shapes in the picture as well.

---

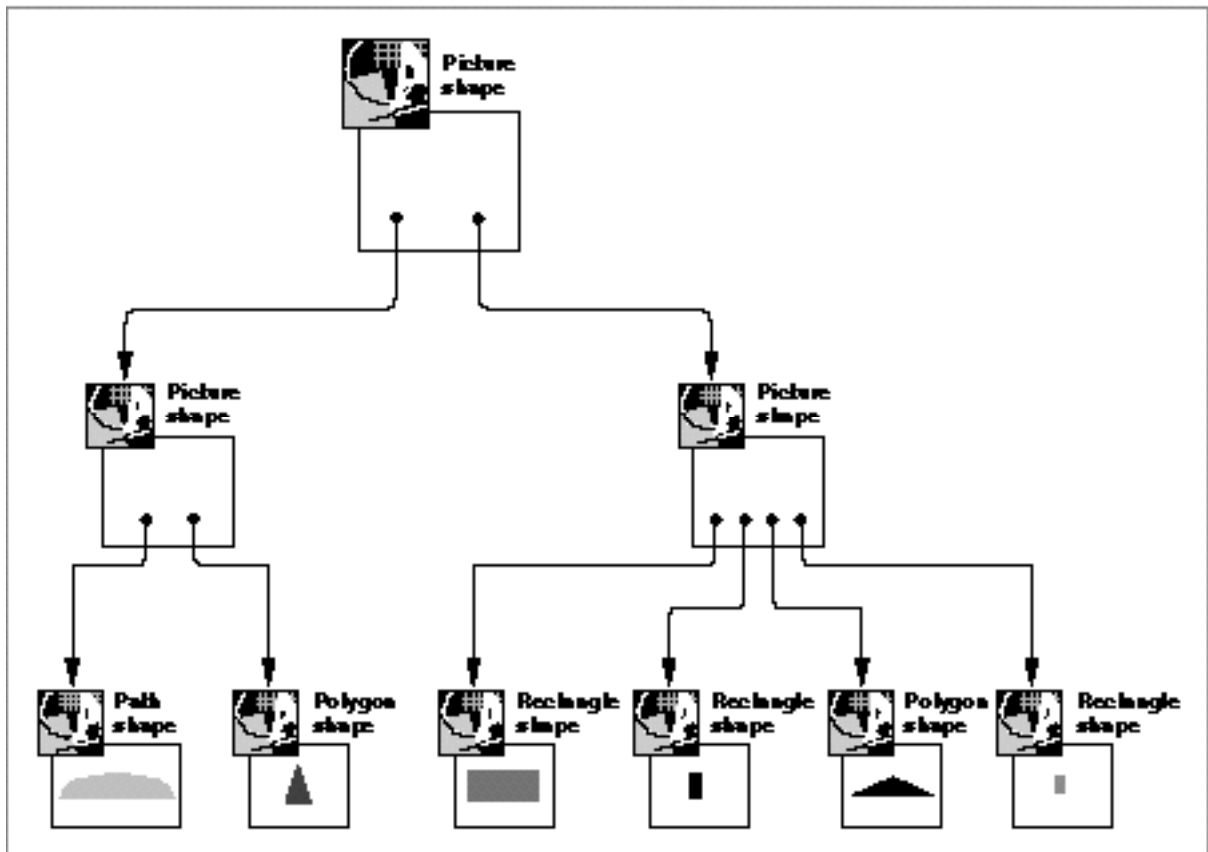
**Figure 1-14** Sample picture shapes



Like bitmap shapes, picture shapes make extensive use of their geometry property. A picture shape uses its geometry property to store a list of references to the shapes to be included the picture. Although each of these shapes has its own style, ink, and transform object, picture shapes allow you to provide an overriding style, ink, and transform object to use for each of these shapes.

Figure 1-15 shows a hierarchical view of the first picture shape shown in Figure 1-14. The picture contains two items: each of which is a picture shape itself. The first item is a picture that contains two items: the lawn and the walkway. The second item is a picture that contains four items: the chimney, the house, the door, and the roof.

**Figure 1-15** A picture hierarchy



Notice that the order the shapes appear in the geometry is the order in which QuickDraw GX draws them, from back to front.

Since picture shapes contain other shapes, they don't make much use of their shape fill property, although you can specify a no-fill shape fill if you don't want the picture to appear when drawn.

Picture shapes also don't make much use of their associated style or ink objects, since each shape in the picture has its own style object and ink object, and, potentially, an overriding style and ink object.

Picture shapes do make full use of their transform objects, however. For example, you can scale, skew, rotate, and clip picture shapes as a whole, as well as separately for each individual shape in the picture. QuickDraw GX also provides powerful tools for hit-testing picture shapes.

For more information about picture shapes, see the chapter "Picture Shapes" in this book.

# Geometric Shapes

---

## Contents

About Geometric Shapes	2-5
The Geometric Properties of Shape Objects	2-7
Shape Type	2-7
Shape Geometry	2-9
Shape Fill	2-12
The Geometric Shape Types	2-16
Empty Shapes and Full Shapes	2-16
Point Shapes	2-16
Line Shapes	2-17
Curve Shapes	2-18
Rectangle Shapes	2-20
Polygon Shapes	2-22
Path Shapes	2-25
Using Geometric Shapes	2-27
Creating and Drawing Empty Shapes and Full Shapes	2-28
Creating and Drawing Points	2-29
Creating and Drawing Lines	2-36
Creating and Drawing Curves	2-41
Creating and Drawing Rectangles	2-43
Creating and Drawing Polygons	2-45
Creating Polygons With a Single Contour	2-46
Creating Polygons With Multiple Contours	2-49
Creating Polygons With Crossed Contours	2-50
Creating and Drawing Paths	2-55
Creating Paths With a Single Contour	2-57
Creating Paths Using Only Off-Curve Points	2-59
Creating Paths With Multiple Contours	2-60
Converting Between Geometric Shape Types	2-65
Converting Shapes to Points, Lines, and Rectangles	2-66

Converting Shapes to Curve Shapes	2-71
Converting Shapes to Polygons and Paths	2-74
Replacing Geometric Points	2-79
Editing Polygon Parts	2-82
Editing Paths Parts	2-91
Editing Shape Parts	2-93
Applying Functions Described Elsewhere to Geometric Shapes	2-100
Shape-Related Functions Applicable to Geometric Shapes	2-100
Other Functions Applicable to Geometric Shapes	2-103
Geometric Shapes Reference	2-103
Data Types	2-104
The Point Structure	2-104
The Line Structure	2-105
The Curve Structure	2-105
The Rectangle Structure	2-106
Polygon Structures	2-106
Path Structures	2-107
Functions	2-108
Creating Geometric Shapes	2-109
GXNewShapeVector	2-109
GXNewPoint	2-111
GXNewLine	2-112
GXNewCurve	2-113
GXNewRectangle	2-114
GXNewPolygons	2-116
GXNewPaths	2-117
Getting and Setting Shape Geometries	2-119
GXSetShapeVector	2-119
GXGetPoint	2-121
GXSetPoint	2-122
GXGetLine	2-123
GXSetLine	2-124
GXGetCurve	2-125
GXSetCurve	2-126
GXGetRectangle	2-127
GXSetRectangle	2-129
GXGetPolygons	2-130
GXSetPolygons	2-131
GXGetPaths	2-132
GXSetPaths	2-133
Editing Shape Geometries	2-135
GXCountShapeContours	2-136
GXCountShapePoints	2-137
GXGetShapeIndex	2-139
GXGetShapePoints	2-140
GXSetShapePoints	2-142
GXGetPolygonParts	2-144

## CHAPTER 2

GXSetPolygonParts	2-145
GXGetPathParts	2-148
GXSetPathParts	2-149
GXGetShapeParts	2-152
GXSetShapeParts	2-154
<b>Drawing Geometric Shapes</b>	<b>2-157</b>
GXDrawPoint	2-158
GXDrawLine	2-158
GXDrawCurve	2-159
GXDrawRectangle	2-160
GXDrawPolygons	2-161
GXDrawPaths	2-162
<b>Summary of Geometric Shapes</b>	<b>2-163</b>
<b>Constants and Data Types</b>	<b>2-163</b>
<b>Functions</b>	<b>2-164</b>





## Geometric Shapes

This chapter describes the geometric shapes. In particular, it shows you how you can

- n define geometries
- n create geometric shapes
- n manipulate their shape type, shape fill, and geometry properties
- n draw the shapes

Before you read this chapter, you should be familiar with some of the information in *Inside Macintosh: QuickDraw GX Objects*. In particular, you should read the chapters “Introduction to QuickDraw GX Objects” and “Shape Objects” in that book.

The next chapter, “Geometric Styles,” discusses the stylistic variations you can apply to geometric shapes.

Chapter 4, “Geometric Operations,” describes the functions QuickDraw GX provides for performing operations on the geometries of geometric shapes—operations such as intersection, union, and so on.

For information about applying colors and transfer modes to geometric shapes, you should read the chapter “Ink Objects” in *Inside Macintosh: QuickDraw GX Objects*.

For information about applying mapping transformations to geometric shapes, clipping geometric shapes, and hit-testing geometric shapes, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## About Geometric Shapes

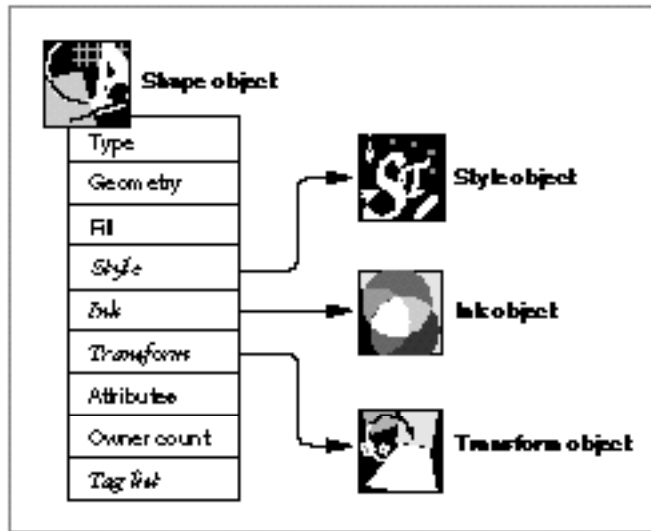
---

QuickDraw GX represents shapes in memory using a shape object and an associated style, ink, and transform object. QuickDraw GX uses these same objects to represent all types of shapes—graphic as well as typographic.

A shape object has nine **properties**, which are like fields of a data structure with one important exception: you cannot directly examine or change the information stored in a property. Instead, you must use QuickDraw GX functions to examine or alter the value of a property.

Figure 2-1 shows a graphic representation of a shape object and its nine properties.

**Figure 2-1** A shape object



The first three properties of a shape object—the shape type, shape geometry, and shape fill—are called the geometric shape properties. These properties are examined in detail in “The Geometric Properties of Shape Objects” beginning on page 2-7. In particular, that section describes how these three properties are used by geometric shapes.

The next three properties of a shape object—the style, ink, and transform properties—are references to the style, ink, and transform objects associated with the shape. Each of these objects contains information that modifies the way QuickDraw GX draws the shape. You can find more information about these objects in *Inside Macintosh: QuickDraw GX Objects*. In addition, you can find specific information about how style objects affect geometric shapes in Chapter 3, “Geometric Styles,” in this book.

The final three properties of a shape object—the shape attributes, the owner count, and the tag list—are the object-related shape properties. You can find information about these properties, and how they affect all types of shapes, in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

QuickDraw GX provides six basic types of geometric shapes and two special types. The six basic types include points, lines, curves, rectangles, polygons, and paths; the two special types include empty shapes and full shapes.

## Geometric Shapes

Each of these shape types is examined in detail in “The Geometric Shape Types” beginning on page 2-16. In particular, that section analyzes how each type of geometric shape uses its shape geometry and shape fill, and also discusses the default geometric shapes.

## The Geometric Properties of Shape Objects

---

Every shape object has three geometric properties: the shape type, the shape geometry, and the shape fill. For geometric shapes, these properties define

- n the type of shape—for example, a point, a line, or a curve
- n the coordinates of the shape—for example, the position where a line starts and ends, or the positions of the corners of a rectangle
- n how the shape is filled—for example, whether the shape is framed (drawn as an outline) or solid (drawn as a solid area)

The next three sections examine these properties in more detail.

### Shape Type

---

The **shape type** property of a shape object specifies what type of shape the shape object represents. There are thirteen different QuickDraw GX shape types: one for bitmap shapes, one for picture shapes, three for typographic shapes, and eight for geometric shapes. The eight geometric shape types are:

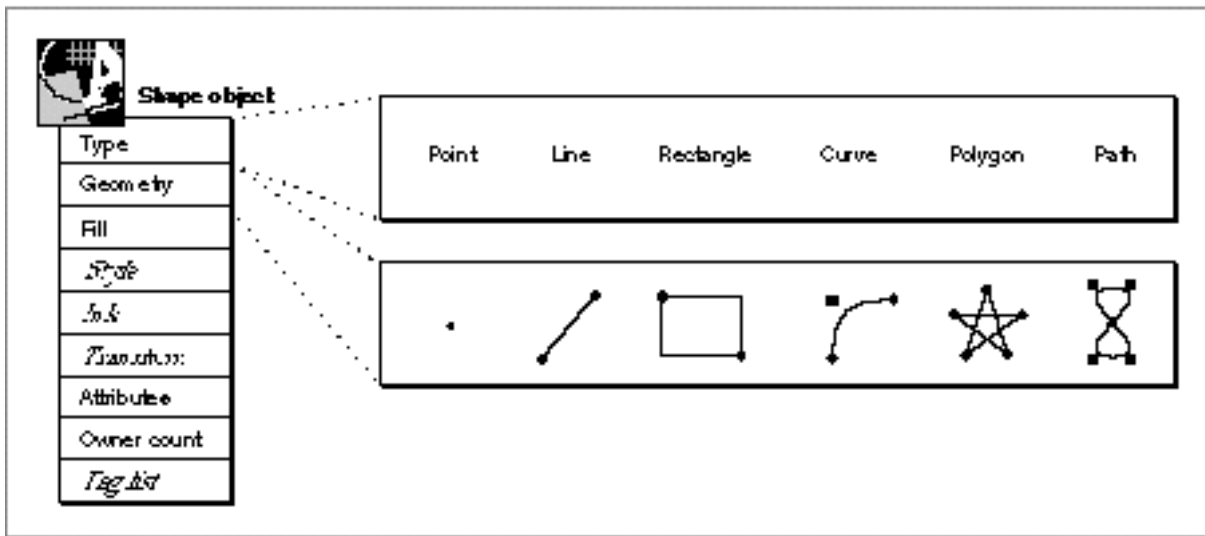
- n point
- n line
- n curve
- n rectangle
- n polygon
- n path
- n empty
- n full

The value of the shape type property affects the way QuickDraw GX interprets the other properties of the shape. In particular, different types of shapes store substantially different information in their geometry properties. For example, the geometry of a point shape contains only an x-coordinate and a y-coordinate. The geometry of a line contains an x-coordinate and a y-coordinate to define the beginning of the line and an x-coordinate and a y-coordinate to define the end of the line. The geometry of a polygon shape can contain many pairs of (x, y) coordinates.

## Geometric Shapes

Figure 2-2 shows a shape object and lists six possible values for its shape type property. This figure also shows a sample geometry for each of the shape types listed. Each geometry is made up of geometric points (specified by (x, y) coordinate pairs) and edges connecting the geometric points. The next section, “Shape Geometry,” discusses geometric points and edges in more detail.

**Figure 2-2** The geometric shape types and examples of geometric shape geometries



There are two types of geometric shapes not shown in this figure: the empty shape and the full shape. An empty shape is a shape that has no geometry and covers no area. A full shape is the inverse of an empty shape—it covers all area. You can find more information about these shape types in “Empty Shapes and Full Shapes” beginning on page 2-16.

## Shape Geometry

---

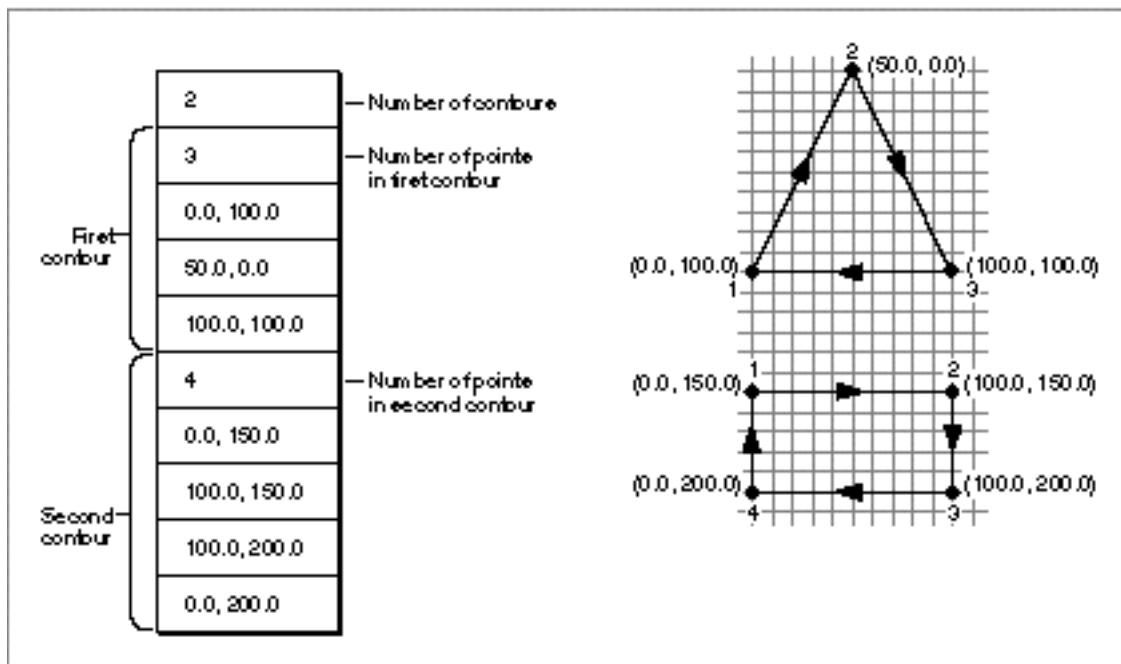
Each type of geometric shape uses the **geometry** property of its shape object in a slightly different manner. For example, empty shapes and full shapes store no information in their geometry, because they require no further geometric information—their shape type says it all.

However, for other types of geometric shapes, the shape type does not contain all the geometric information necessary to define the shape. The geometries of these shapes contain (x, y) coordinate pairs called **geometric points**—points that specify the location, dimension, and form of the geometric shapes:

- <sup>n</sup> Point geometries contain one geometric point—an x-coordinate and a y-coordinate—to specify the position of the point shape. See “Point Shapes” on page 2-16 for more information.
- <sup>n</sup> Line geometries contain two geometric points—one point to specify where the line starts and one to specify where the line ends. See “Line Shapes” on page 2-17 for more information.
- <sup>n</sup> Rectangle geometries also contain two geometric points—specifying the positions of opposing corners of the rectangle. See “Rectangle Shapes” on page 2-20 for more information.
- <sup>n</sup> Curve shapes store three geometric points in their geometry—one to specify where the curve starts, another to specify where the curve ends, and another, called the **off-curve control point**, to specify the tangents used to define the curve. See “Curve Shapes” on page 2-18 for more information.
- <sup>n</sup> A polygon shape can contain multiple contours. A **polygon contour** is a series of geometric points connected by straight lines—for example, a V-shape, a triangle, or a hexagon.
- <sup>n</sup> A path geometry can also contain multiple contours, but each **path contour** can contain curves as well as straight lines.

Figure 2-3 shows a polygon shape with a two polygon contours made up of seven geometric points total. This figure shows two views of the polygon geometry: as a list of (x, y) coordinate pairs and as seven geometric points plotted on a geometric grid. This second way of viewing geometries is used frequently throughout this book, as it shows not only the geometric points, but also the implied **edges** that connect them. Typically, the figures in this book do not show the grid, but just the points and edges.

**Figure 2-3** A polygon shape with a single contour containing three geometric points



## Geometric Shapes

Each geometric point in a geometry has a **geometry index**—if you consider the geometry as a list of geometric points starting from the first geometric point of the first contour to the last geometric point of the last contour, the geometry index of a particular geometric point is its position in this list. For example, in the shape in Figure 2-3, the first point (0.0, 100.0) has a geometry index of 1, the second point (50.0, 0.0) has a geometry index of 2, and the third point (100.0, 100.0) has a geometry index of 3. The first point in the second contour (0.0, 150.0) has a geometry index of 4, as it is the fourth geometric point in the geometry. However, it has a contour index of 1, as it is the first point of its contour. Similarly, the next point (100.0, 150.0) has a geometry index of 5 and a contour index of 2, and so forth.

Notice that each of the three edges of the polygon contour in Figure 2-3 has a direction. The first edge is pointing up and to the right; the second edge is pointing down and to the right; the third edge is pointing to the left. QuickDraw GX takes into consideration the direction that an edge is pointing in a number of circumstances:

- n When filling a shape. QuickDraw GX allows you to choose how a shape should be filled. The next section, “Shape Fill,” discusses how the direction of an edge can affect how QuickDraw GX fills a shape.
- n When determining the **contour direction** of a contour. In the example in Figure 2-3, both polygon contours have a clockwise contour direction. If their geometric points were reversed, the polygon contours would have a counterclockwise contour direction.
- n When determining the inside or outside of a contour. QuickDraw GX normally defines the right side of an edge to be the inside and the left side to be the outside. Since the example in Figure 2-3 has a clockwise contour direction, the inside of the contour corresponds to what you would expect the inside to be. If the contour had a counterclockwise direction, the inside of the contour would correspond to what you might expect the outside to be.

QuickDraw GX uses contour direction and the inside and outside of a shape when applying certain stylistic variations, as described in Chapter 3, “Geometric Styles,” and when performing certain geometric operations, as described in Chapter 4, “Geometric Operations,” of this book.

For more details about the geometries of the various geometric shapes, see “The Geometric Shape Types” beginning on page 2-16.

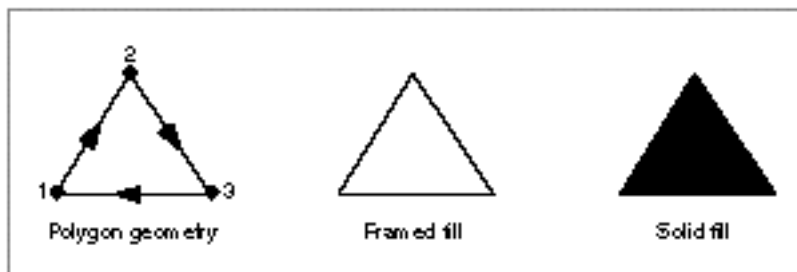
## Shape Fill

The shape fill property specifies how QuickDraw GX interprets the geometric points of a geometric shape's geometry during drawing and other operations. There are two basic types of shape fills:

- n **Framed fills.** These shape fills indicate that QuickDraw GX should interpret the shape as an outline—as a series of edges.
- n **Solid fills.** These shape fills indicate that QuickDraw GX should interpret the shape as a solid area—the edges of the shape represent the boundaries of the area.

Figure 2-4 shows an example of a polygon contour similar to the one in Figure 2-3, and how QuickDraw GX might draw it with a framed fill and with a solid fill.

**Figure 2-4** Framed shapes versus solid shapes



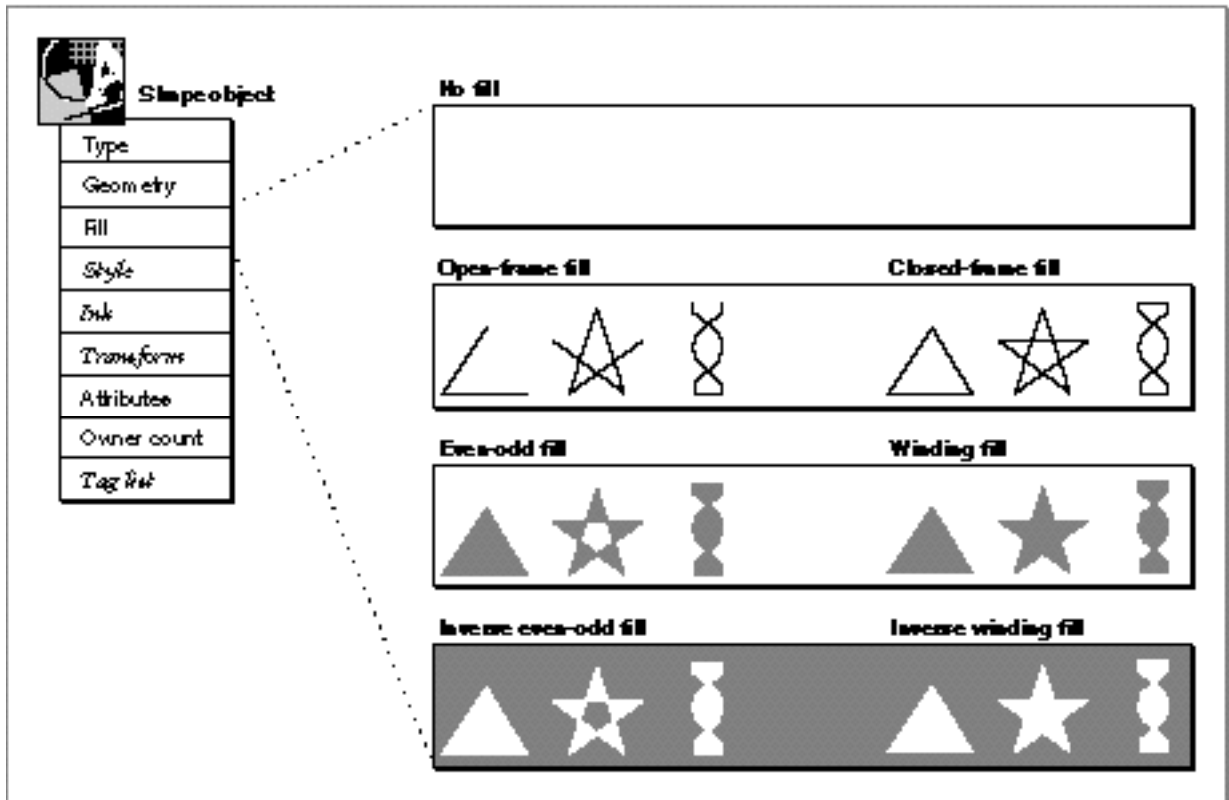
QuickDraw GX actually provides seven types of shape fills:

- n no-fill shape fill
- n open-frame shape fill (also called *frame fill*)
- n closed-frame shape fill (also called *hollow fill*)
- n even-odd shape fill (also called *solid fill*)
- n winding shape fill
- n inverse even-odd shape fill (also called *inverse fill* and *inverse solid fill*)
- n inverse winding shape fill



Figure 2-5 shows these shape fills and the effect they have on three sample geometries.

**Figure 2-5** The various shape fills and examples of their effects



The no-fill shape fill specifies that QuickDraw GX should not draw the shape. You can use this shape fill to hide a shape. You can specify the no-fill shape fill for any shape type.

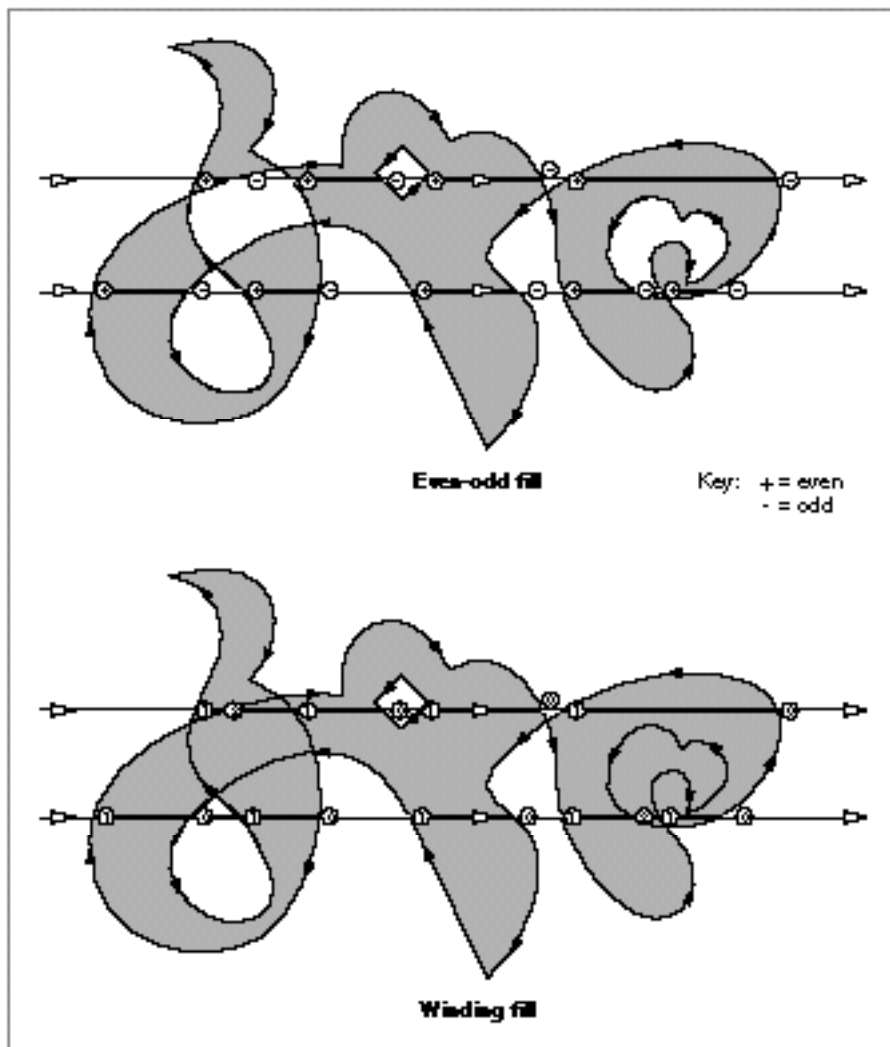
The open-frame shape fill specifies that QuickDraw GX should draw a shape as a connected set of edges. The closed-frame shape fill indicates that QuickDraw GX should also connect the last geometric point of a contour to the first geometric point of that contour.

The even-odd shape fill and the winding shape fill indicate that QuickDraw GX should interpret the shape as a solid area—the edges of the shape represent the boundaries of the area. These two shape fills differ in the algorithm they use to determine what area to include in the shape.

The even-odd shape fill indicates that QuickDraw GX should use the **even-odd rule** to determine what area lies inside a shape. As QuickDraw GX scans a shape horizontally, it fills the area between every other pair of edges, as shown in Figure 2-6.

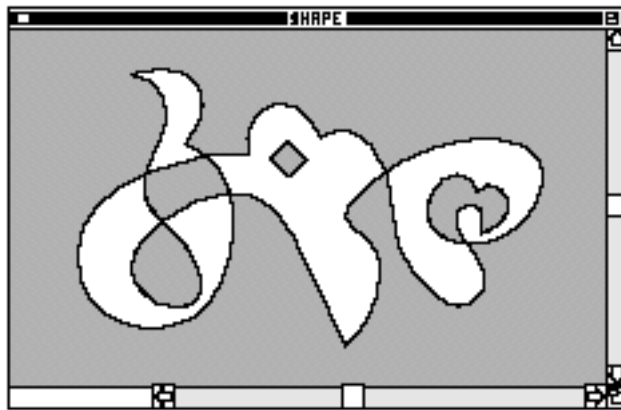
The winding shape fill indicates that QuickDraw GX should use the **winding-number rule** to determine what area lies inside a shape. As QuickDraw GX scans a shape horizontally, it increments a counter the first time it crosses an edge of the shape. It also notices whether the contour was directed up or down at that edge. As QuickDraw GX continues to scan the shape horizontally, everytime it crosses another edge pointed in the same direction (up or down), it increments the counter, and when it crosses an edge pointing in the opposite direction (down or up), it decrements the counter. Wherever along the horizontal scan line the counter is not zero, QuickDraw GX fills the area, as is shown in Figure 2-6.

**Figure 2-6** The even-odd rule and winding-number rule algorithms



The inverse even-odd shape fill indicates the inverse of the even-odd shape fill, as shown in Figure 2-7.

**Figure 2-7** The inverse even-odd shape fill



Similarly, the inverse winding shape fill indicates the inverse of the winding shape fill.

Not all shape fills are appropriate for all types of geometric shapes. For example, a rectangle shape can have a closed-frame shape fill but not an open-frame shape fill; a line shape can only have an no-fill or an open-frame shape fill.

See the sections on each shape type, beginning on page 2-16, for a complete discussion of the shape fills that are allowed for each shape type.

The shape fill does more than affect the way a shape is drawn; it affects the fundamental behavior of a shape. Two shapes with the same geometry that have different shape fills can exhibit vastly different geometric behaviors. For example, the shape fill can affect

- n stylistic variations, which are described in Chapter 3, “Geometric Styles,” in this book
- n shape measurements and other geometric operations, which are discussed in Chapter 4, “Geometric Operations,” in this book (As an example, a polygon with the closed-frame shape fill might simplify to a rectangle. However, the same polygon with the open-frame fill might not simplify at all.)
- n hit-testing, which is described in the chapter “Transform Objects” and the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*

For examples of how shape fill affects the behavior of shapes, see

- n “Polygon Shapes” beginning on page 2-22
- n “Path Shapes” beginning on page 2-25
- n “Creating and Drawing Polygons” beginning on page 2-45
- n “Creating and Drawing Paths” beginning on page 2-55

## The Geometric Shape Types

---

QuickDraw GX provides eight types of geometric shapes: empty shapes, full shapes, point shapes, line shapes, curve shapes, rectangle shapes, polygon shapes, and path shapes.

The following sections examine each of these shape types in detail. In particular, these sections discuss how the different types of shapes use their geometry and shape fill properties, and what the default values are for properties of each type of shape.

### Empty Shapes and Full Shapes

---

Empty shapes and full shapes are the only geometric shapes with no information stored in the geometry property.

An **empty shape** is a shape with no geometry. When you draw an empty shape, nothing appears. You can use an empty shape when creating other types of shapes. For example, you can create an empty shape and then build it into a polygon shape, adding one contour at a time.

A **full shape** is a shape that covers the largest area possible. When you draw a full shape, QuickDraw GX fills in the entire drawable area of the full shape's view port (paying attention to the clipping information stored in the full shape's transform). You can use a full shape when erasing an area.

### Point Shapes

---

The **point shape** is the simplest of the geometric shapes. Its geometry consists of a single geometric point—a single (x, y) coordinate pair.

Point shapes must always have the open-frame shape fill or the no-fill shape fill.

A point shape's style determines how QuickDraw GX draws the point. If a point's style has a pen width of 0, which is the default pen width, QuickDraw GX draws the point as a single pixel on the output device. If the style has a pen width greater than 0, QuickDraw GX draws the point only if the style also has a start cap. The next chapter, "Geometric Styles," discusses these aspects of the style object in more detail.

When you create a new point shape, QuickDraw GX makes a copy of the default point shape. The default point shape has these properties:

- n owner count: 1
- n tag list: no tags
- n shape attributes: no attributes
- n shape type: point type
- n shape fill: open-frame fill
- n geometry: (0.0, 0.0)

You may change the properties of the default point shape, which effectively changes the behavior of the functions that create point shapes. However, when creating a new point shape, QuickDraw GX always initializes the owner count to 1 and the geometry to (0.0, 0.0), even if you have specified other values for the default point shape.

For examples of creating and drawing point shapes, see “Creating and Drawing Points” beginning on page 2-29.

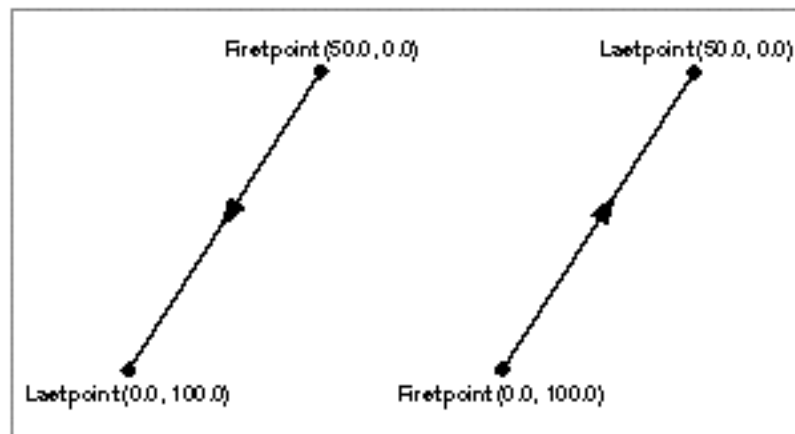
## Line Shapes

The geometry of a **line shape** consists of two geometric points: a first point and a last point. Because the points are ordered, a line points in a certain direction.

Line shapes must always have the open-frame shape fill or the no-fill shape fill.

Figure 2-8 shows two line shapes. The geometries of these two lines have the same geometric points, but in the opposite order. Therefore, the two lines point in opposite directions.

**Figure 2-8** Two lines



If a line shape uses the default style information, the direction of the line does not affect how QuickDraw GX draws the line. However, when you add stylistic variations (such as pen width, pen placement, and dashes) to a line shape, the direction of the line can affect how QuickDraw GX draws the line. See the next chapter, “Geometric Styles,” for information about how you can add stylistic variations to a line.

When you create a new line shape, QuickDraw GX makes a copy of the default line shape. The default line shape has these properties:

- n owner count: 1
- n tag list: no tags
- n shape attributes: no attributes

## Geometric Shapes

- n shape type: line type
- n shape fill: open-frame fill
- n geometry: (0.0, 0.0), (0.0, 0.0)

You may change the properties of the default line shape, which effectively changes the behavior of the functions that create line shapes. However, when creating a new line shape, QuickDraw GX always initializes the owner count to 1 and the geometry to (0.0, 0.0), (0.0, 0.0), even if you have specified other values for the default line shape.

For examples of creating and drawing line shapes without stylistic variations, see “Creating and Drawing Lines” beginning on page 2-36.

For examples of creating and drawing lines with stylistic variations, see the next chapter, “Geometric Styles.”

## Curve Shapes

---

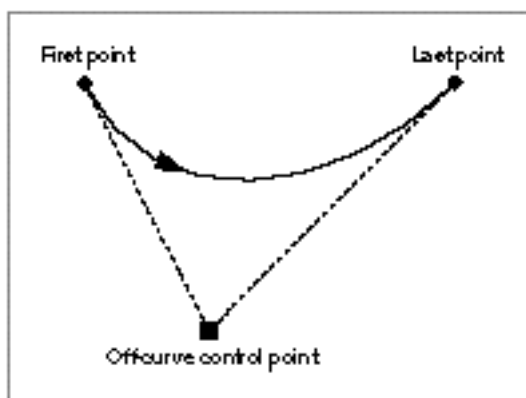
The geometry of a **curve shape** consists of three geometric points: a first point, a last point, and an off-curve control point that determines the tangents of the curve. The curve described by these three points is a quadratic Bézier curve—the same type of curve used to describe TrueType fonts.

Because a curve’s geometric points are ordered, a curve has direction. As with line shapes, direction affects the drawing of a curve only after you apply stylistic variations, which are discussed in the next chapter, “Geometric Styles.”

Curve shapes must always have the open-frame shape fill or no fill shape fill.

Figure 2-9 shows an example of a curve shape. In this example, the first point is (50.0, 50.0), the last point is (200.0, 50.0) and the off-curve control point is (100.0, 150.0).

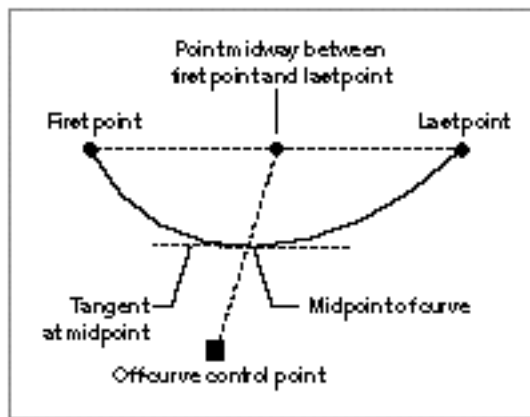
**Figure 2-9** A quadratic Bézier curve



Quadratic Bézier curves have the following characteristics:

- n A line connecting the first point and the off-curve control point describes the tangent of the curve at the first point.
- n A line connecting the off-curve control point and the last point describes the tangent of the curve at the last point.
- n The curve is always contained by the triangle formed by the three geometric points.
- n The midpoint of the curve is halfway between the off-curve control point and the point midway between the first point and last point, as shown in Figure 2-10.

**Figure 2-10** Finding the midpoint of a curve

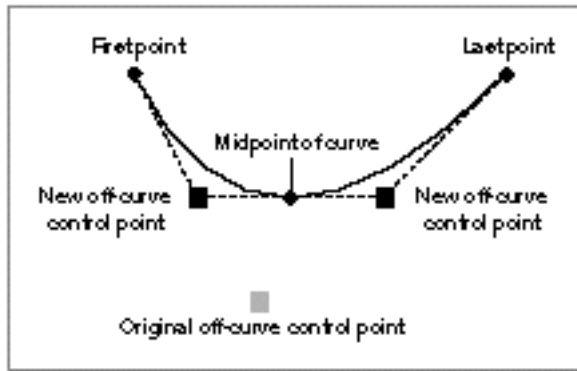


You can divide a quadratic Bézier curve into two smaller quadratic Bézier curves:

- n One smaller curve extends from the first point to the midpoint of the original curve. The new off-curve control point is the point midway between the first point and the original off-curve control point.
- n The other smaller curve extends from the midpoint to the last point of the original curve. The new off-curve control point is the point midway between the original off-curve control point and the last point.

Figure 2-11 shows a curve divided into two smaller curves.

**Figure 2-11** Dividing a curve into two smaller curves



When you create a new curve shape, QuickDraw GX makes a copy of the default curve shape. The default curve shape has these properties:

- n owner count: 1
- n tag list: no tags
- n shape attributes: no attributes
- n shape type: curve type
- n shape fill: open-frame fill
- n geometry: (0.0, 0.0), (0.0, 0.0), (0.0, 0.0)

You may change the properties of the default curve shape, which effectively changes the behavior of the functions that create curve shapes. However, when creating a new curve shape, QuickDraw GX always initializes the owner count to 1 and the geometry to (0.0, 0.0), (0.0, 0.0), (0.0, 0.0), even if you have specified other values for the default curve shape.

For examples of creating and drawing curve shapes without stylistic variations, see “Creating and Drawing Curves” beginning on page 2-41.

For examples of creating and drawing curves with stylistic variations, see the next chapter, “Geometric Styles.”

## Rectangle Shapes

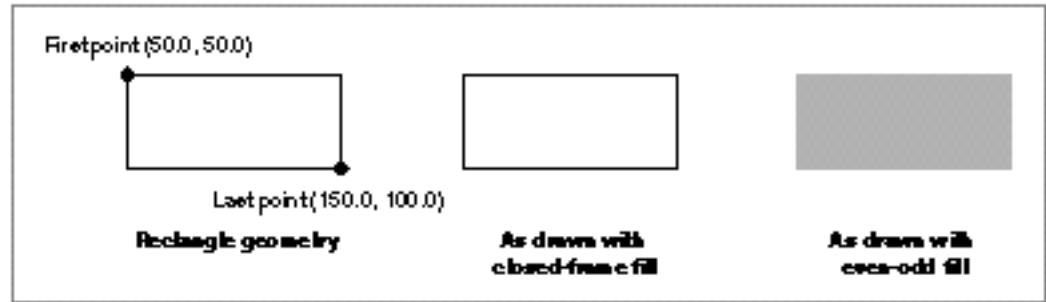
The geometry of a **rectangle shape** consists of two geometric points. Typically, these geometric points represent the upper-left and lower-right corners of the rectangle; however, you can specify any corner as the first geometric point and the diagonally opposite corner as the second geometric point.



Rectangle shapes can have any shape fill except the open-frame shape fill.

Figure 2-12 shows a rectangle geometry and how that rectangle is drawn with a closed-frame shape fill and how it is drawn with an even-odd shape fill.

**Figure 2-12** A rectangle geometry shown framed and filled



#### Note

Although you may specify a rectangle's geometric points in any order, QuickDraw GX functions that calculate rectangles always return rectangles with the upper-left corner as the first geometric point and the lower-right corner as the second geometric point.  $\square$

When you create a new rectangle shape, QuickDraw GX makes a copy of the default rectangle shape. The default rectangle shape has these properties:

- n owner count: 1
- n tag list: no tags
- n shape attributes: no attributes
- n shape type: rectangle type
- n shape fill: even-odd fill
- n geometry: (0.0, 0.0), (0.0, 0.0)

You may change the properties of the default rectangle shape, which effectively changes the behavior of the functions that create rectangle shapes. However, when creating a new rectangle shape, QuickDraw GX always initializes the owner count to 1 and the geometry to (0.0, 0.0), (0.0, 0.0), even if you have specified other values for the default rectangle shape.

For examples of creating and drawing rectangle shapes without stylistic variations, see "Creating and Drawing Rectangles" beginning on page 2-43.

For examples of creating and drawing rectangles with stylistic variations, see the next chapter, "Geometric Styles."

## Polygon Shapes

---

A **polygon contour** is a series of geometric points connected by straight lines. A **polygon shape** may include any number of polygon contours.

### Implementation Note

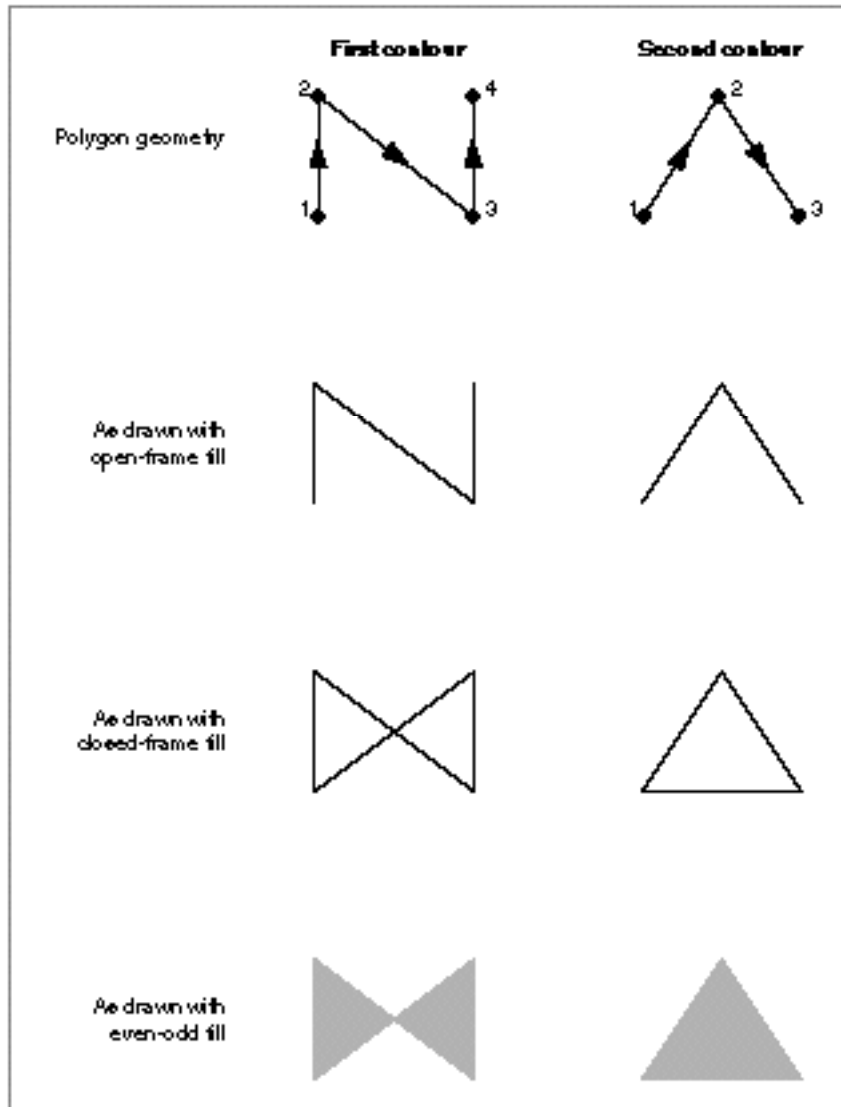
In version 1.0 of QuickDraw GX, a single polygon contour can have between 1 and 32,767 geometric points. The geometry of a polygon shape can have between 0 and 32,767 polygon contours. The total size of a polygon geometry may not exceed 2,147,483,647 bytes. u

Polygon shapes may have any shape fill.

Figure 2-13 shows a polygon shape that contains two separate contours. The shape is shown four times:

- n as a polygon shape geometry
- n as drawn with the open-frame shape fill
- n as drawn with the closed-frame shape fill
- n as drawn with even-odd shape fill

**Figure 2-13** A polygon shape with two polygon contours



The first contour in Figure 2-13 has four geometric points and the second contour has three geometric points.

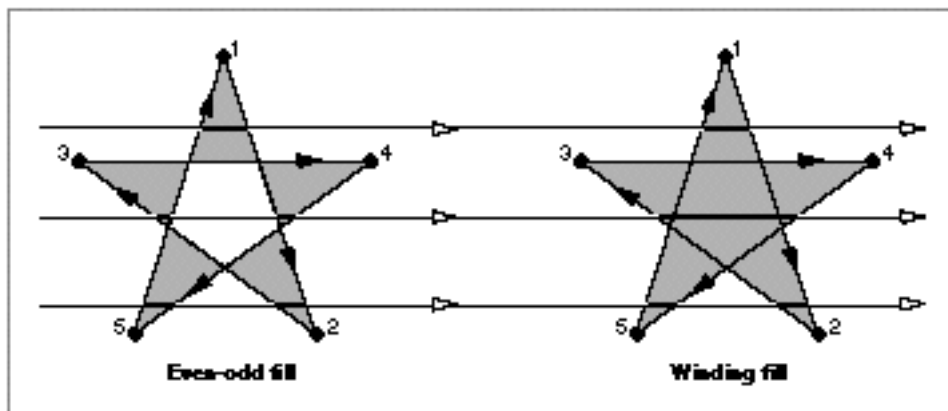
## Geometric Shapes

The index of a geometric point within its contour is called its **contour index**. The geometric points in the first contour in Figure 2-13 have contour indexes ranging from 1 to 4, and the geometric points in the second contour in Figure 2-13 have contour indexes ranging from 1 to 3. These contour indexes are shown in the top part of the figure.

Since contours and geometric points are ordered, each geometric point can be numbered from the first geometric point of the first contour to the last geometric point of the last contour. This number is called a geometric point's **geometry index**. Since the polygon geometry in Figure 2-13 has seven geometric points total, these points have geometry indexes ranging from 1 to 7. You use geometry indexes and contour indexes of geometric points when editing polygon geometries. For examples, see "Editing Polygon Parts" beginning on page 2-82.

If the contours of a polygon shape cross over one another, or if a polygon shape contains contours that lie within other contours, the even-odd shape fill and the winding shape fill may fill the polygon shape differently, as shown in Figure 2-14.

**Figure 2-14** A polygon drawn with the even-odd and winding shape fills



When you create a new polygon shape, QuickDraw GX makes a copy of the default polygon shape. The default polygon shape has these properties:

- n owner count: 1
- n tag list: no tags
- n shape attributes: no attributes
- n shape type: polygon type
- n shape fill: even-odd fill
- n geometry: 0 contours, 0 points

You may change the properties of the default polygon shape, which effectively changes the behavior of the functions that create polygon shapes. However, when creating a new polygon shape, QuickDraw GX always initializes the owner count to 1 and the geometry to 0 contours with 0 points, even if you have specified other values for the default polygon shape.

For examples of creating and drawing polygon shapes without stylistic variations, see “Creating and Drawing Polygons” beginning on page 2-45.

For examples of creating and drawing polygons with stylistic variations, see the next chapter, “Geometric Styles.”

## Path Shapes

A **path contour**, like a polygon contour, is defined by a series of geometric points. However, a path contour can contain off-curve control points as well as on-curve points; therefore, a path contour can contain curves as well as straight lines. A **path shape** may include any number of path contours.

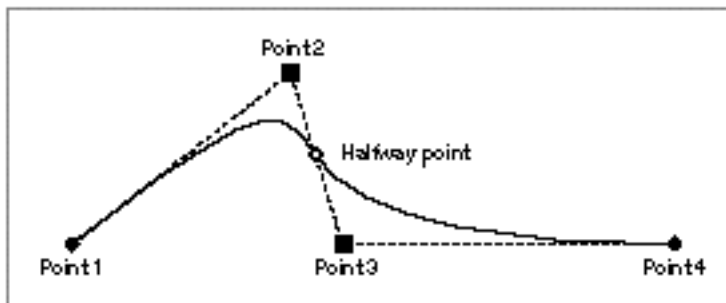
### Implementation Note

In version 1.0 of QuickDraw GX, a single path contour can have between 0 and 32,767 geometric points. The geometry of a path shape can between 0 and 32,767 polygon contours. The total size of a path geometry may not exceed 2,147,483,647 bytes. u

Every path contains an array of **control bits** that specify which geometric points are on curve and which geometric points are off curve. QuickDraw GX connects two consecutive on-curve points with a straight line. If two on-curve points have an off-curve point between them, QuickDraw GX connects the two on-curve points with a quadratic Bézier curve, using the geometric point between them as the off-curve control point.

QuickDraw GX allows a path to have two or more consecutive off-curve control points. In this case, each pair of consecutive off-curve points implies an on-curve point midway between them, as represented by the small hollow circle in Figure 2-15.

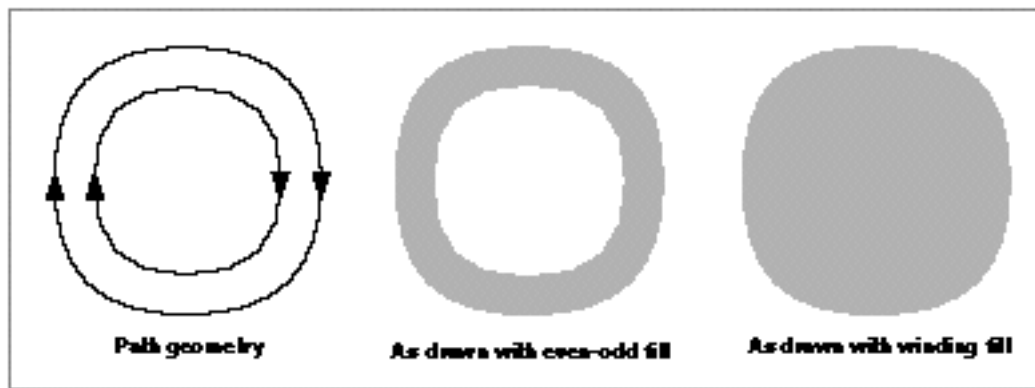
**Figure 2-15** A path with two consecutive off-curve points



Path shapes may have any shape fill—including open-frame shape fill. However, a path may not have the open-frame shape fill if the first point or the last point of any path contour is an off-curve point.

If the contours of a path shape cross over one another, or if a path shape contains contours that lie within other contours, the even-odd shape fill and the winding shape fill may fill the path shape differently, as shown in Figure 2-16.

**Figure 2-16** A path shape filled with the even-odd and winding shape fills



Contour direction affects how QuickDraw GX fills a path when the path has the winding shape fill. In the example in Figure 2-16, if the inner contour has the opposite contour direction from the outer contour, the winding shape fill works in the same manner as the even-odd shape fill. For more information, see the next section, “Shape Fill.” For examples, see “Creating and Drawing Paths” beginning on page 2-55.

When you create a new path shape, QuickDraw GX makes a copy of the default path shape. The default path shape has these properties:

- n owner count: 1
- n tag list: no tags
- n shape attributes: no attributes
- n shape type: path type
- n shape fill: even-odd fill
- n geometry: 0 contours, 0 points

You may change the properties of the default path shape, which effectively changes the behavior of the functions that create path shapes. However, when creating a new path shape, QuickDraw GX always initializes the owner count to 1 and the geometry to 0 contours with 0 points, even if you have specified other values for the default paths shape.

For examples of creating and drawing path shapes without stylistic variations, see “Creating and Drawing Paths” beginning on page 2-55.

For examples of creating and drawing paths with stylistic variations, see the next chapter, “Geometric Styles.”

## Using Geometric Shapes

---

This section shows you how to create, edit, and draw geometric shapes. In particular, this section shows you how to

- n create and draw empty and full shapes
- n create point, line, curve, rectangle, polygon, and path shapes
- n draw points, lines, curves, rectangles, polygons, and paths
- n create framed and solid shapes
- n convert a shape from one shape type to another
- n replace the geometry of a shape
- n replace geometric points within a shape's geometry
- n insert geometric points into and remove geometric points from a shape's geometry

All of the sample functions in this section create geometric shapes with default style, ink, and transform information. All shapes are black; framed shapes have one-pixel-wide contours; and the shapes are not rotated, skewed, and so on. For examples of the many stylistic variations you can apply to geometric shapes, see Chapter 3 of this book, “Geometric Styles.” For information about inks and transforms, see *Inside Macintosh: QuickDraw GX Objects*.

Many of the sample functions in this section create geometric shapes and, to do so, they specify geometric points for the shapes' geometries. Since a geometric point contains two fixed-point values (of type `Fixed`), the sample functions in this section must convert integer constants to fixed-point constants when specifying a geometric point.

QuickDraw GX provides the `GXIntToFixed` macro, which performs this conversion by shifting the integer value 16 bits to the left:

```
#define GXIntToFixed(a) ((Fixed)(a) << 16)
```

## Geometric Shapes

QuickDraw GX also provides the `ff` macro as a convenient alias:

```
#define ff(a) GXIntToFixed(a)
```

A few of the sample functions in this section specify fractional values for geometric point coordinates. To convert a floating-point value (of type `float`) to a fixed-point value (type `Fixed`), QuickDraw GX provides the `GXFloatToFixed` macro:

```
#define GXFloatToFixed(a) ((Fixed)((float)(a) * fixed1))
```

and the synonymous `fl` macro:

```
#define fl(a) GXFloatToFixed(a)
```

**IMPORTANT**

The `GXIntToFixed` macro has substantially faster performance than the `GXFloatToFixed` macro. Whenever possible, you should choose the `GXIntToFixed` macro over the `GXFloatToFixed` macro. *s*

## Creating and Drawing Empty Shapes and Full Shapes

---

To create an empty shape or a full shape, you use the function `GXNewShape`, which is described in full in the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects*.

To create an empty shape, you could define a shape reference and then call the `GXNewShape` function:

```
gxShape anEmptyShape;
```

```
anEmptyShape = GXNewShape(gxEmptyType);
```

Although you can draw this shape with the `GXDrawShape` function, nothing will appear. However, you can use empty shapes for other purposes. For example, you can create an empty shape and then add geometric points to it using the `SetShapeParts` function, building other types of shapes as you add points. See “Editing Shape Parts” beginning on page 2-93 for examples of this function.



## Geometric Shapes

To create a full shape, you can use this code:

```
gxShape aFullShape;

aFullShape = GXNewShape(gxFullType);
```

You can then draw the full shape to cover the entire area of the shape's view ports. For example, you could use the full shape to erase an area, or you could set the color of the full shape and draw it to create a colored background before drawing other shapes.

## Creating and Drawing Points

---

QuickDraw GX provides a number of methods to create and draw geometric shapes. In general, to draw a shape you must first define a geometry. You can then draw the shape in one of two ways:

- n You can draw the geometry directly—without having to create a shape object.
- n You can create a shape object to encapsulate the geometry and then draw the shape.

The first sample function in this section, shown in Listing 2-1, uses the first method—it draws a point without creating a point shape.

To draw the point, this sample function first defines a point geometry, which is represented by a point structure (of type `gxPoint`):

```
struct gxPoint {
    Fixed    x;
    Fixed    y;
};
```

The value in the `x` field specifies horizontal distance from the origin; greater values indicate distances further to the right. The value in the `y` field specifies vertical distance from the origin; greater values indicate distances further down.

### Note

The coordinates of a shape's geometry go through a number of transformations before the shape is actually drawn. Where the shape is drawn depends not only on the values of the shape's geometry, but also on the shape's associated transform and view port objects. If you use the default transform and view port information, the coordinates in a shape's geometry represent units of 1/72 inch and the origin is the upper-left corner of the view port. See *Inside Macintosh: QuickDraw GX Objects* for more information about the coordinate systems of QuickDraw GX. u

## Geometric Shapes

Since each coordinate of a point must be a fixed-point value, the sample function in Listing 2-1 uses the `GXIntToFixed` macro to convert integer constants to fixed-point constants.

The sample function then draws the point using the `GXDrawPoint` function. The `GXDrawPoint` function takes a pointer to a `gxPoint` structure as its only parameter and draws the corresponding point. When drawing the point, it uses the style, ink, and transform information associated with the default point shape.

---

**Listing 2-1** Drawing a point without creating a point shape

```
void DrawASinglePoint(void)
{
    static gxPoint aPointGeometry = {GXIntToFixed(5),
                                     GXIntToFixed(5)};

    GXDrawPoint(&aPointGeometry);
}
```

QuickDraw GX provides the `ff` macro as an alias for the `GXIntToFixed` macro. In the example in Listing 2-1, the point coordinates could be specified with this line of code:

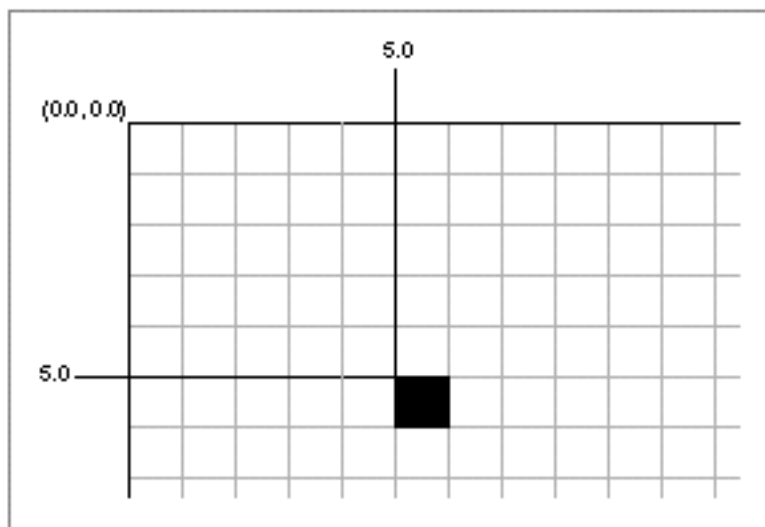
```
static gxPoint aPointGeometry = {ff(5), ff(5)};
```

The rest of the examples in “Using Geometric Shapes” use this convenient alternative.

Figure 2-17 shows the result of the sample function from Listing 2-1.

---

**Figure 2-17** A point



Listing 2-1 defines the point at location (5.0, 5.0), which lies at the intersection of two infinitely thin grid lines, and therefore is infinitely thin itself. However, when QuickDraw GX draws this point shape, it draws it as a single pixel—the pixel lying down and to the right of the point itself, as shown in Figure 2-17. QuickDraw GX only draws this single-pixel type of point, called a **hairline point**, if the pen width property of the style object associated with the point shape has a value of 0, which is the default value for this property. If the pen width is greater than 0.0, QuickDraw GX does not draw the point, unless it has a start cap, in which case only the start cap is drawn. For more information about the pen width property and cap property of style objects and how they affects the drawing of point shapes, see the chapter “Geometric Styles,” in this book.

Although you may sometimes want to draw a shape without creating a shape object for it, you will frequently want to create a shape object before drawing a shape. Creating a shape object has many advantages; for example, it allows you to provide custom style, ink, and transform information before drawing the shape.

QuickDraw GX provides three main methods for creating geometric shapes:

- n You can call a type-specific function, such as `GXNewPoint`, which requires you to provide a pointer to the shape’s desired geometric structure.
- n You can call the `GXNewShapeVector` function, which requires you to specify the shape type and provide a pointer to the shape’s desired geometric structure.
- n You can call the `GXNewShape` function, which requires you to specify the desired shape type, and then call a type-specific function, such as `GXSetPoint`, to set the geometry.

The sample functions in Listing 2-2, Listing 2-3, and Listing 2-4 show how to create a point shape using these three methods.

Listing 2-2 uses the `GXNewPoint` function to create a point shape given a pointer to a point geometry.

---

**Listing 2-2**      Creating a point shape with the `GXNewPoint` function

```
void CreatePointShape(void)
{
    gxShape      aPointShape;
    static gxPoint aPointGeometry = {ff(5), ff(5)};

    aPointShape = GXNewPoint(&aPointGeometry);

    GXDrawShape(aPointShape);

    GXDisposeShape(aPointShape);
}
```

## Geometric Shapes

**Listing 2-3** uses the `GXNewShapeVector` function to create a point shape. The `GXNewShapeVector` function requires two parameters:

- n the shape type of the shape you want to create
- n an array of fixed-point values that represent the shape's geometry

In this example, the desired shape type is `gxPointType` and the geometry is specified as an array of two fixed-point values representing the coordinates of the point's geometry. When using the `GXNewShapeVector` function to create shapes more complicated than point shapes, you need to provide more values in this array.

---

**Listing 2-3** Creating a point shape with the `GXNewShapeVector` function

```
void CreatePointShape(void)
{
    gxShape      aPointShape;
    static Fixed aPointGeometry[] = {ff(5), ff(5)};

    aPointShape = GXNewShapeVector(gxPointType, aPointGeometry);

    GXDrawShape(aPointShape);

    GXDisposeShape(aPointShape);
}
```

**Listing 2-4** creates a point shape using the `GXNewShape` function. The `GXNewShape` function requires only that you specify the type of shape to create. You do not have to specify any values for the geometric points of the shape's geometry—the `GXNewShape` function initializes the point geometry to (0.0, 0.0).

## Geometric Shapes

To set the values of the point shape's geometry once it's created, the sample function in Listing 2-4 uses the `GXSetPoint` function. This function takes a reference to the shape and a pointer to the desired geometry as its parameters.

---

**Listing 2-4** Creating a point shape with the `GXNewShape` and `GXSetPoint` functions

```
void CreatePointShape(void)
{
    gxShape aPointShape;

    static gxPoint aPointGeometry = {ff(5), ff(5)};

    aPointShape = GXNewShape(gxPointType);
    GXSetPoint(aPointShape, &aPointGeometry);

    GXDrawShape(aPointShape);

    GXDisposeShape(aPointShape);
}
```

The sample functions in Listing 2-2, Listing 2-3, and Listing 2-4 all use the `GXDrawShape` function to draw the point after the point shape has been created. The resulting point is the same for all three examples; it appears as shown in Figure 2-17.

## Geometric Shapes

You can use the `GXSetPoint` function to replace a point shape's geometry any number of times. The sample function in Listing 2-5 creates a point shape, sets its geometry using the `GXSetPoint` function, draws the point, replaces its geometry using the `GXSetPoint` function, and draws the point again.

---

**Listing 2-5** Using the `GXSetPoint` function to replace a point shape's geometry

```
void ReplacePointShapeGeometry(void)
{
    gxShape  aPointShape;

    static gxPoint aPointGeometry = {ff(5), ff(5)};
    static gxPoint anotherPointGeometry = {ff(13), ff(8)};

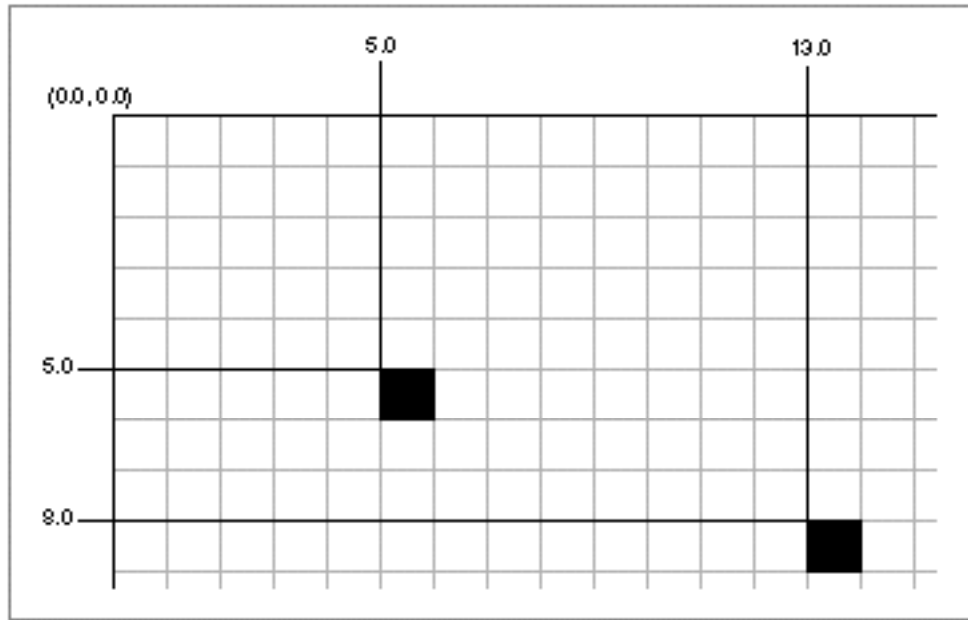
    aPointShape = GXNewShape(gxPointType);
    GXSetPoint(aPointShape, &aPointGeometry);
    GXDrawShape(aPointShape);

    GXSetPoint(aPointShape, &anotherPointGeometry);
    GXDrawShape(aPointShape);

    GXDisposeShape(aPointShape);
}
```

Figure 2-18 depicts the results of this sample function.

**Figure 2-18** Two different point geometries



## Geometric Shapes

Most of the sample functions discussed in “Using Geometric Shapes” create shape objects. If you create a shape object using any of the methods discussed, you are responsible for disposing of the shape when you no longer need it. You can do this using the `GXDisposeShape` function, which decrements the owner count of the shape and frees the memory occupied by that shape if the shape’s owner count becomes 0. The examples of this section dispose of the point shape by calling

```
GXDisposeShape(aPointShape);
```

Since the `GXNewPoint`, `GXNewShapeVector`, and `GXNewShape` functions all return a shape with an owner count of 1, calling the `GXDisposeShape` function in the three previous examples would decrement the owner count to 0 and therefore purge the point shape from memory. For a complete discussion of creating and disposing of shapes, see *Inside Macintosh: QuickDraw GX Objects*.

For more information about point shapes, see “Point Shapes” on page 2-16 and “The Point Structure” on page 2-104.

For information about the functions you can use to create and draw points, see the description of the `GXNewPoint` function on page 2-111 and the `GXDrawPoint` function on page 2-158.

## Creating and Drawing Lines

---

You can draw lines and create line shapes with QuickDraw GX in much the same way as you draw points and create point shapes. Typically, you first define a line geometry, which is encapsulated in a `gxLine` structure:

```
struct gxLine {  
    struct gxPoint first;  
    struct gxPoint last;  
};
```



Once you've defined a line geometry, you can draw the corresponding line without creating a line shape by using the `GXDrawLine` function, as shown in Listing 2-6.

---

**Listing 2-6** Drawing a line without creating a line shape

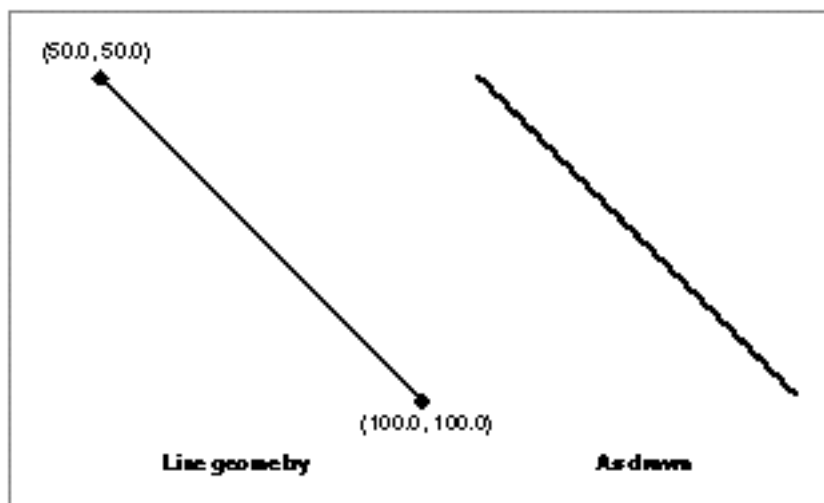
```
void DrawASingleLine(void)
{
    static gxLine aLineGeometry = {{ff(50), ff(50)},
                                    {ff(150), ff(150)}};

    GXDrawLine(&aLineGeometry);
}
```

This sample function defines a line geometry, using the `ff` macro (which is an alias for the `GXIntToFixed` macro) to convert integer constants to fixed-point coordinate values. It then uses the `GXDrawLine` function to draw the line. The `GXDrawLine` function uses the style, ink, and transform information from the default line shape when drawing the line. The result is shown in Figure 2-19.

---

**Figure 2-19** A line



## Geometric Shapes

As with the point shape in Figure 2-17, the line shape in Figure 2-19 is infinitely thin, but is drawn one-pixel wide—a hairline—because the default value of the pen width property of the style object is 0, which indicates that QuickDraw GX should draw the line at the thinnest perceivable resolution.

Another method of drawing a line is to encapsulate the line geometry in a line shape and then use the `GXDrawShape` function to draw the line. This method allows you to specify different style, ink, and transform information for the line. The sample function in Listing 2-7 uses this method: it creates a line shape using the `GXNewLine` function and then draws the line using the `GXDrawShape` function.

---

**Listing 2-7**      Creating a line shape with the `GXNewLine` function

```
void CreateLineShape(void)
{
    gxShape      aLineShape;

    static gxLine aLineGeometry = {ff(50), ff(50),
                                    ff(150), ff(150)};

    aLineShape = GXNewLine(&aLineGeometry);

    GXDrawShape(aLineShape);

    GXDisposeShape(aLineShape);
}
```

You can also use the `GXNewShape` or `GXNewShapeVector` functions to create a line shape. For example, to create the same line shape using the `GXNewShape` function, you could replace this line of code in the previous example:

```
aLineShape = GXNewLine(&aLineGeometry);
```

with these lines of code:

```
aLineShape = GXNewShape(gxLineType);
GXSetLine(aLineShape, &aLineGeometry);
```

In either case, the line shape would be the same, and would appear as shown in Figure 2-19.

The sample function in Listing 2-8 shows how to use the `GXSetLine` function to change the geometry of an existing line shape.

**Listing 2-8** Drawing two parallel lines

---

```

void DrawParallelLines(void)
{
    gxShape      aLineShape;

    static gxLine aLineGeometry = {ff(50), ff(50),
                                    ff(57), ff(100)};

    static gxLine anotherLineGeometry = {ff(60), ff(50),
                                          ff(67), ff(100)};

    aLineShape = GXNewShape(gxLineType);
    GXSetLine(aLineShape, &aLineGeometry);

    GXDrawShape(aLineShape);

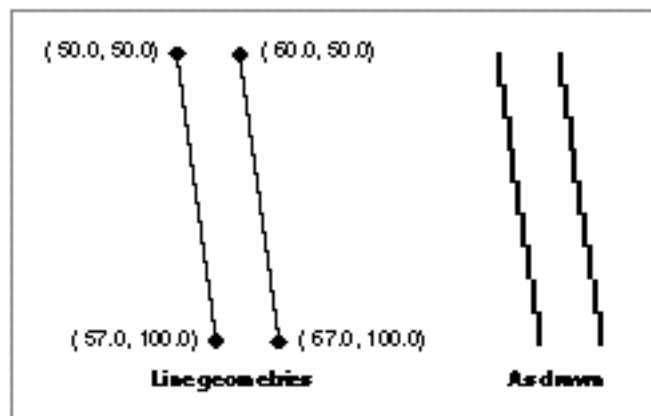
    GXSetLine(aLineShape, &anotherLineGeometry);

    GXDrawShape(aLineShape);

    GXDisposeShape(aLineShape);
}

```

This sample function creates and draws a line shape, changes its geometry, and then draws it again. The results are shown in Figure 2-20.

**Figure 2-20** Parallel lines

As with any geometric shape, you can specify fractional values for a line shape's geometric points. Although specifying a fractional part does not move the start pixel or the end pixel of line (unless rounding occurs), it can affect how the line is drawn. When QuickDraw GX draws a line with fractional endpoint coordinates, rather than integer endpoint coordinates, it may choose different pixels to represent the line, even if the endpoints remain on the same pixels in both cases. By choosing a different “stair step” pattern to represent the line, QuickDraw GX can give the illusion of very slight changes in line angles. As an example, if in the previous example you replace the second definition:

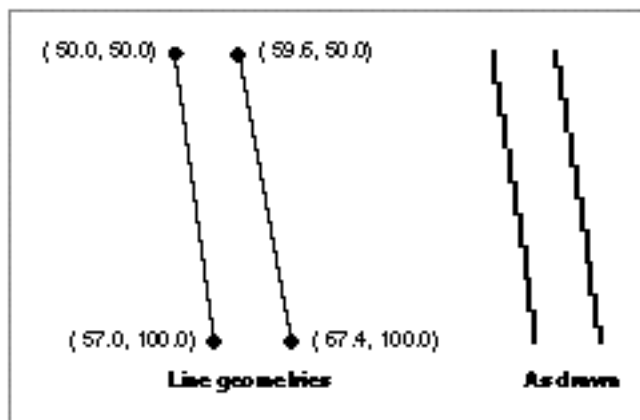
```
static gxLine anotherLineGeometry = {ff(60), ff(50),
                                     ff(67), ff(100)};
```

with a slightly modified version:

```
static gxLine anotherLineGeometry = {fl(59.6), ff(50),
                                     fl(67.4), ff(100)};
```

QuickDraw GX chooses different pixels to represent the second line, giving the appearance of a slightly different angle, as shown in Figure 2-21.

**Figure 2-21** Nearly parallel lines



For more information about line shapes, see “Line Shapes” on page 2-17 and “The Line Structure” on page 2-105.

For information about the functions you can use to create and draw lines, see the description of the `GXNewLine` function on page 2-112 and the `GXDrawLine` function on page 2-158.

## Creating and Drawing Curves

---

You can create and draw curve shapes with QuickDraw GX the same way you create and draw points and lines. Typically, you first define a curve geometry, which is encapsulated in a `gxCurve` structure:

```
struct gxCurve {
    struct gxPoint first;
    struct gxPoint control;
    struct gxPoint last;
};
```

The `first` and `last` fields determine the start point and the end point of the curve. The point specified in the `control` field lies off the curve and determines the tangents of the curve. (The off-curve control point could actually be on the curve—that is, directly between the first and last points—in which case the curve is a straight line.)

Once you've defined a curve geometry, you can create a curve shape using the `GXNewCurve` function and draw it using the `GXDrawShape` function, as shown in Listing 2-9.

---

**Listing 2-9**      Creating a curve shape

```
void CreateCurve(void)
{
    gxShape aCurveShape;

    static gxCurve aCurveGeometry = {ff(50), ff(50),    /* on */
                                     ff(100), ff(150), /* off */
                                     ff(200), ff(50)}; /* on */

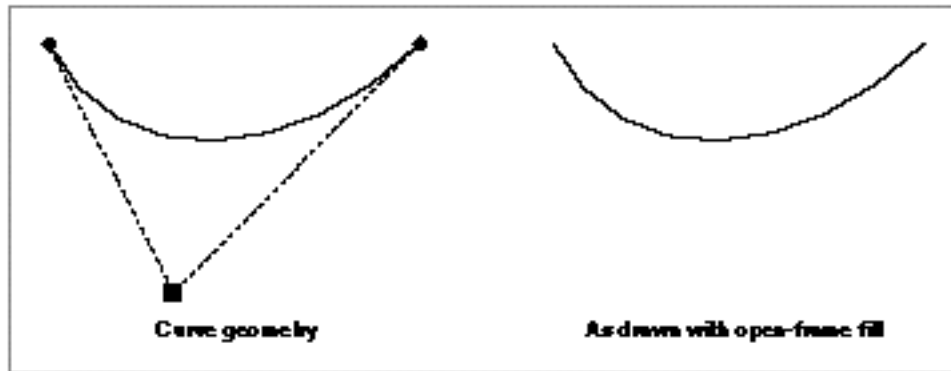
    aCurveShape = GXNewCurve(&aCurveGeometry);

    GXDrawShape(aCurveShape);

    GXDisposeShape(aCurveShape);
}
```

Figure 2-22 shows the curve shape geometry, which includes the first and last points, the off-curve control point, and the tangents implied by these geometric points. This figure also shows the curve as drawn. It is drawn as a hairline (one-pixel wide) with the open-frame shape fill, which reflects the default values for curve shapes.

**Figure 2-22** A curve



You could draw the same curve without creating a curve shape by calling the `GXDrawCurve` function:

```
GXDrawCurve(&aCurveGeometry);
```

You could also create the curve shape using the `GXNewShape` function described in *Inside Macintosh: QuickDraw GX Objects* or the `GXNewShapeVector` function described on page 2-109.

Curves have a direction that depends on the order of the points in the geometry. For example, you could reverse the direction of the curve in Figure 2-22 by reversing the order of the points in the geometry definition from Listing 2-9:

```
static gxCurve aCurveGeometry = {ff(200), ff(50), /* on curve */
                                ff(100), ff(150), /* off curve */
                                ff(50), ff(50)}; /* on curve */
```

Changing the direction of this curve would not change its appearance. However, curve direction can affect the appearance of a curve when you apply certain stylistic variations, such as dashing, to the curve. The next chapter, “Geometric Styles,” discusses these stylistic variations. Also, when a curve is part of a path shape, the direction of the curve can affect the way the path is drawn. See “Creating and Drawing Paths” beginning on page 2-55 for examples of how the direction of a curve can affect drawing.

For more information about curve shapes, see “Curve Shapes” on page 2-18 and “The Curve Structure” on page 2-105. For information about the functions you can use to create and draw curves, see the description of the `GXNewCurve` function on page 2-113 and the `GXDrawCurve` function on page 2-159.

## Creating and Drawing Rectangles

---

You can create rectangle shapes and draw rectangles with QuickDraw GX the same way you create and draw points, lines, and curves. Typically, you first define a rectangle geometry, which is encapsulated in a `gxRectangle` structure:

```
struct gxRectangle {
    Fixed    left;
    Fixed    top;
    Fixed    right;
    Fixed    bottom;
};
```

### Note

QuickDraw GX allows you to specify rectangle coordinates out of order—that is, you can specify any corner of the rectangle using the first two fields of the rectangle structure and the opposing corner using the third and fourth fields of the rectangle structure. <sup>u</sup>

Once you've defined a rectangle geometry, you can draw the corresponding rectangle without creating a rectangle shape by using the `GXDrawRectangle` function or you can create a rectangle shape and draw it with the `GXDrawShape` function, as shown in Listing 2-10.

---

**Listing 2-10**    Creating a rectangle shape

```
void CreateRectangle(void)
{
    gxShape aRectangleShape;

    static gxRectangle aRectangleGeometry = {ff(50), ff(50),
                                              ff(150), ff(100)};

    aRectangleShape = GXNewRectangle(&aRectangleGeometry);

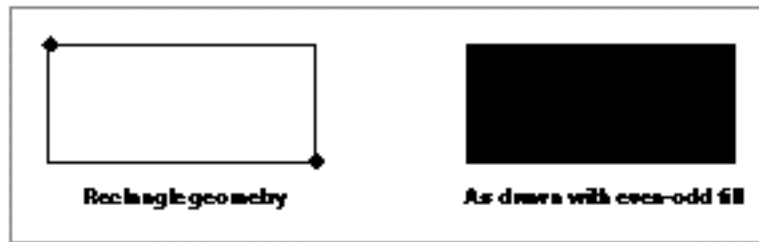
    GXDrawShape(aRectangleShape);

    GXDisposeShape(aRectangleShape);
}
```

This sample function uses the `ff` macro (which is an alias for the `IntegerToFixed` macro) to convert integer constants to the fixed-point coordinate values needed to define a rectangle geometry. It then creates a rectangle shape using the `GXNewRectangle` function (although it could use the `GXNewShape` function or

the `GXNewShapeVector` function instead) and draws the rectangle using the `GXDrawShape` function. The result is shown in Figure 2-23.

**Figure 2-23** A rectangle



Notice that the rectangle is solid rather than framed. The `GXNewRectangle` function returns a new rectangle shape with the same shape fill property as the default rectangle shape, which is the even-odd shape fill.

#### Note

Although initially QuickDraw GX sets the shape fill property of the default rectangle shape to be even-odd shape fill, you may change the default shape fill for rectangles by using the `GXGetDefaultShape` function to obtain a reference to the default rectangle and then using the `GXSetShapeFill` function to change its shape fill. You can similarly change the default shape fill for any shape type. ▮

To create a framed rectangle, you can use the `GXSetShapeFill` function to change the shape fill from even-odd to closed-frame, as shown in Listing 2-11.

**Listing 2-11** Creating a framed rectangle

```
void CreateFramedRectangle(void)
{
    gxShape aRectangleShape;

    static gxRectangle aRectangleGeometry = {ff(150), ff(100),
                                             ff(50), ff(50)};

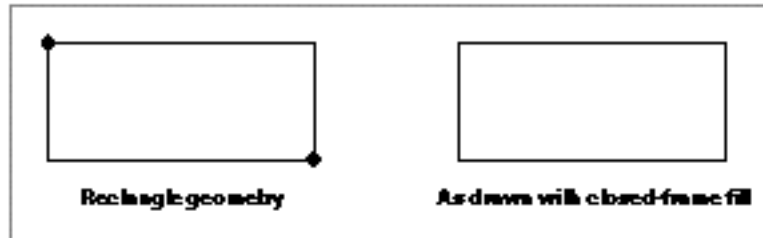
    aRectangleShape = GXNewRectangle(&aRectangleGeometry);
    GXSetShapeFill(aRectangleShape, gxClosedFrameFill);

    GXDrawShape(aRectangleShape);
}
```



Figure 2-24 shows the result of Listing 2-11.

**Figure 2-24** A framed rectangle



In general, a rectangle can have any shape fill except open-frame shape fill. For more information about rectangle shapes, see “Rectangle Shapes” on page 2-20 and “The Rectangle Structure” on page 2-106.

For more information about the shape fill property, see “Shape Fill” beginning on page 2-12.

For information about the functions you can use to create and draw rectangles, see the description of the `GXNewRectangle` function on page 2-114 and the `GXDrawRectangle` function on page 2-160.

## Creating and Drawing Polygons

A polygon contour is a series of points connected by straight lines. QuickDraw GX defines the `gxPolygon` structure to encapsulate a polygon contour:

```
struct gxPolygon {
    long          vectors;
    struct gxPoint vector[gxAnyNumber];
};
```

The `vectors` field indicates the number of points in the polygon and the `vector` array contains the points themselves. (The constant `gxAnyNumber` is used as a placeholder, since a polygon contour can have any number of geometric points.)

The polygon shape type allows you to group any number of polygon contours within a single QuickDraw GX shape. The `gxPolygons` structure encapsulates the multiple-polygon geometry:

```
struct gxPolygons {
    long          contours;
    struct gxPolygon contour[gxAnyNumber];
};
```

## Geometric Shapes

The `contours` field indicates the total number of contours (in other words, the total number of separate polygons), and the `contour` array contains the polygon contour geometries.

**Implementation Note**

In version 1.0 of QuickDraw GX, a single path contour can have between 0 and 32,767 geometric points. The geometry of a path shape can between 0 and 32,767 polygon contours. The total size of a path geometry may not exceed 2,147,483,647 bytes. <sup>u</sup>

## Creating Polygons With a Single Contour

---

Since a `gxPolygons` structure is of variable length and every element in it is of type `long`, you can define a polygon geometry as an array of `long` values. For example, the definition

```
long  aPolygonGeometry[] = {1, /* number of contours */
                             3, /* number of points */
                             ff(50), ff(50),
                             ff(100), ff(80),
                             ff(50), ff(110)};
```

defines a polygon geometry with one contour (that is, with one polygon). The polygon contains three points; it is a triangle.

Most QuickDraw GX functions that create or draw polygon shapes expect a pointer to a `gxPolygons` structure as one of the parameters. Therefore, you must cast an array of `long` values to the correct type before sending it to one of these functions. As an example, you can cast the `aPolygonGeometry` array to the correct type with this expression:

```
(gxPolygons *) aPolygonGeometry
```

The sample function in Listing 2-12 shows how to use this geometry to draw a triangle.

---

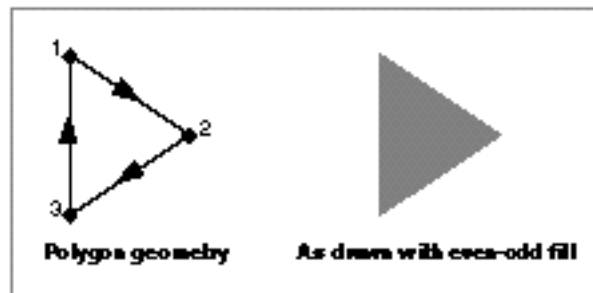
**Listing 2-12** Drawing a triangular polygon

```
void DrawTriangle(void)
{
    static long aPolygonGeometry[] = {1, /* number of contours */
                                       3, /* number of points */
                                       ff(50), ff(50),
                                       ff(100), ff(80),
                                       ff(50), ff(110)};

    GXDrawPolygons((gxPolygons *) aPolygonGeometry, gxEvenOddFill);
}
```

This sample function defines the `aPolygonGeometry` array, casts it to a `gxPolygons` pointer, and sends it to the `GXDrawPolygons` function. Unlike the `GXDrawPoint`, `GXDrawLine`, `GXDrawCurve`, and `GXDrawRectangle` functions, the `GXDrawPolygons` function takes a second parameter—the shape fill to use when drawing the polygon shape. In this example, the parameter is set to the even-odd shape value and the resulting polygon is shown in Figure 2-25.

**Figure 2-25** A polygon

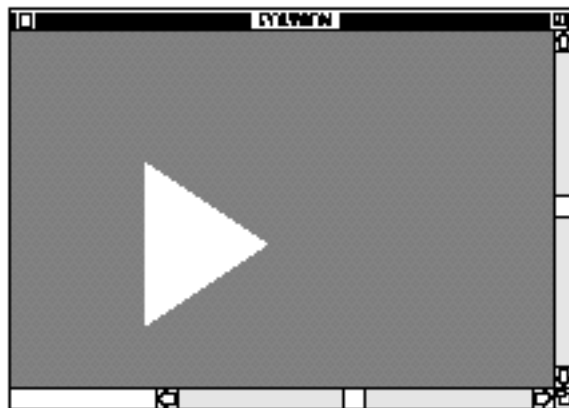


You can specify any type of shape fill for polygon shapes. For example, if you specify the inverse even-odd shape fill:

```
GXDrawPolygons((gxPolygons *) aPolygonGeometry,  
               gxInverseEvenOddFill);
```

QuickDraw GX draws the graphic shown in Figure 2-26. The black portion of the drawing would be clipped according to the information in the default polygon shape's transform object. If no clipping information is specified there, the drawing would extend to the full range of the shape's view port.

**Figure 2-26** A triangular polygon with inverse shape fill



## Geometric Shapes

For information on clipping and view ports, see *Inside Macintosh: QuickDraw GX Objects*.

Although this example draws the polygon without creating a polygon shape, it could instead create a polygon shape with the `GXNewPolygons` function:

```
aPolygonShape = GXNewPolygons((gxPolygons *) aPolygonGeometry);
```

and then draw it using the `GXDrawShape` function:

```
GXDrawShape(aPolygonShape);
```

You can also create polygon shapes using the `GXNewShape` function:

```
aPolygonShape = GXNewShape(gxPolygonType);
GXSetPolygons(aPolygonShape, (gxPolygons *) aPolygonGeometry);
```

or by using the `GXNewShapeVector` function:

```
aPolygonShape = GXNewShapeVector(gxPolygonType, aPolygonGeometry);
```

Notice that in this case you do not have to cast the `aPolygonGeometry` array to be a pointer to a `gxPolygons` structure. The `GXNewShapeVector` function expects an array of long values.

Although the `GXDrawPolygons` function (shown in Listing 2-12) allows you to specify a shape fill, the `GXDrawShape` function does not. If you create a polygon shape and you want it to have a different shape fill than the default polygon shape, you must indicate the desired shape fill using the `GXSetShapeFill` function—for example,

```
GXSetShapeFill(aPolygonShape, gxInverseEvenOddFill);
```

For more information about shape fills, see “Shape Fill” beginning on page 2-12.

### Creating Polygons With Multiple Contours

---

The sample function in Listing 2-13 shows how a single polygon shape can contain more than one polygon contour. The polygon shape defined in this example includes the triangle from the previous example as well as a second, entirely separate, triangle.

---

**Listing 2-13** Creating a polygon with two contours

```
void DrawTwoTriangles(void)
{
    gxShape      aPolygonsShape;

    static long aPolygonsGeometry[] = {2, /* number of contours */
                                        3, /* number of points */
                                        ff(50), ff(50),
                                        ff(100), ff(80),
                                        ff(50), ff(110),
                                        3, /* number of points */
                                        ff(200), ff(50),
                                        ff(150), ff(80),
                                        ff(200), ff(110)};

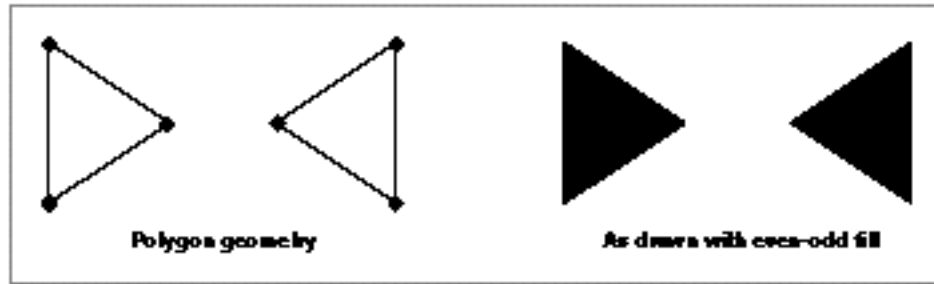
    aPolygonsShape = GXNewPolygons((gxPolygons *)
                                    aPolygonsGeometry);

    GXDrawShape(aPolygonsShape);

    GXDisposeShape(aPolygonsShape);
}
```

This sample function results in the drawing shown in Figure 2-27.

**Figure 2-27** A filled polygon with two separate contours



For more information about polygon shapes and multiple contours, see “Polygon Shapes” beginning on page 2-22.

### Creating Polygons With Crossed Contours

Since a polygon contour is defined as an array of geometric points connected by straight lines, it is possible for the lines that make up a polygon contour to cross over each other. The sample function in Listing 2-14 creates such a polygon.

**Listing 2-14** Creating a polygon with a crossed contour

```
void CreateCrossedContour(void)
{
    gxShape      aPolygonsShape;

    static long aPolygonsGeometry[] = {1, /* number of contours */
                                        4, /* number of points */
                                        ff(50), ff(50),
                                        ff(150), ff(110),
                                        ff(150), ff(50),
                                        ff(50), ff(110)};

    aPolygonsShape = GXNewPolygons((gxPolygons *)
                                    aPolygonsGeometry);
    GXSetShapeFill(aPolygonsShape, gxClosedFrameFill);
}
```

## Geometric Shapes

```

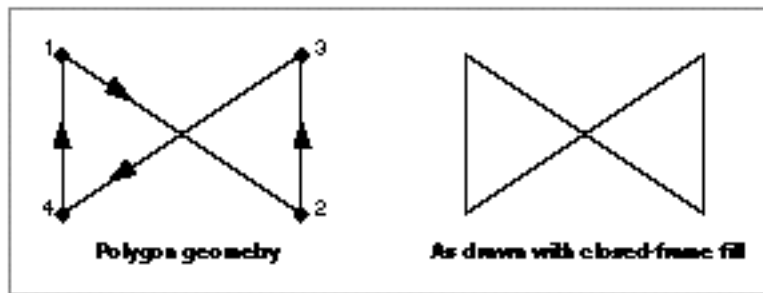
GXDrawShape(aPolygonsShape);

GXDisposeShape(aPolygonsShape);
}

```

Figure 2-28 shows the geometry of the resulting polygon contour as well as how the contour appears when drawn with the closed-frame shape fill.

**Figure 2-28** A framed polygon with a crossed contour



You can change the shape fill of this polygon by removing this line of code from the sample function in Listing 2-14:

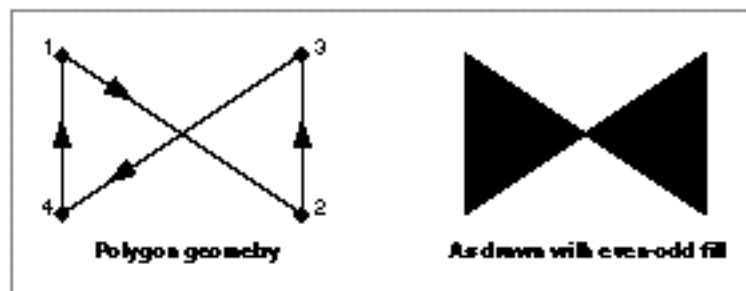
```

GXSetShapeFill(aPolygonsShape, gxClosedFrameFill);

```

If you don't specify a shape fill, the `GXNewPolygons` function uses the shape fill from the default polygon, which is the even-odd shape fill (unless you change it using the `GXGetDefaultShape` and `GXSetShapeFill` functions). The polygon resulting from an even-odd shape fill is shown in Figure 2-29.

**Figure 2-29** A solid polygon with a crossed contour



Notice that QuickDraw GX fills both sections of this polygon.

## Geometric Shapes

It is possible to create a polygon with a contour that overlaps in such a way that QuickDraw GX does not fill all sections of the polygon. The sample function in Listing 2-15 creates such a polygon.

---

**Listing 2-15**    Creating a polygon with an overlapping contour

```
void CreateOverlappingContour(void)
{
    gxShape      aPolygonShape;

    static long aPolygonGeometry[] = {1, /* number of contours */
                                       6, /* number of points */
                                       ff(50), ff(50),
                                       ff(100), ff(80),
                                       ff(25), ff(150),
                                       ff(25), ff(10),
                                       ff(100), ff(80),
                                       ff(50), ff(110)};

    aPolygonShape = GXNewPolygons((gxPolygons *) aPolygonGeometry);
    GXSetShapeFill(aPolygonShape, gxHollowFill);

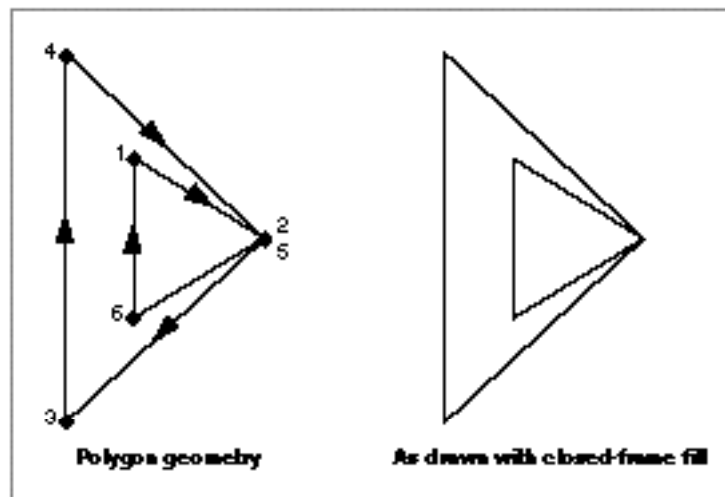
    GXDrawShape(aPolygonShape);

    GXDisposeShape(aPolygonShape);
}
```



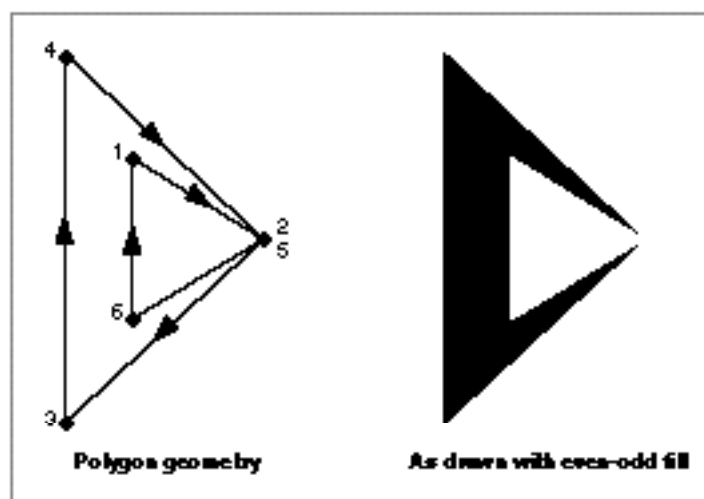
Figure 2-30 shows the geometry of the resulting polygon contour as well as how the contour appears when drawn with the closed-frame shape fill.

**Figure 2-30** A polygon with an overlapping contour and closed-frame shape fill



If you specified the even-odd shape fill for this polygon, instead of the closed-frame shape fill, the resulting shape would appear as in Figure 2-31.

**Figure 2-31** A polygon with an overlapping contour and even-odd shape fill



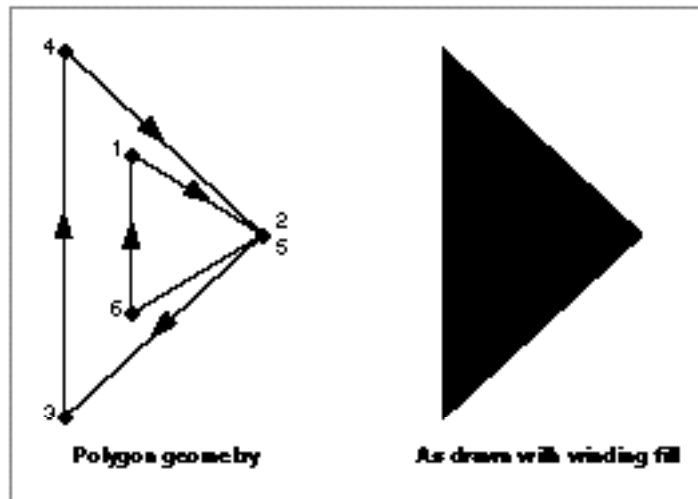
Notice that QuickDraw GX fills in the polygon but does not fill in the area contained in the inner loop. The algorithm used by QuickDraw GX to fill in shapes with the even-odd shape fill doesn't fill loops within the shape. (It would, however, fill another loop inside the first loop.)

The winding shape fill works differently. If you specify the winding shape fill for this polygon using the call

```
GXSetShapeFill(aPolygonShape, gxWindingFill);
```

QuickDraw GX draws the polygon as shown in Figure 2-32.

**Figure 2-32** A polygon with an overlapping contour and winding shape fill



As you can see, the winding shape fill causes QuickDraw GX to hide the inner loop—it fills in the entire polygon, outer loop and inner.

It is possible, however, to define a polygon in such a way that QuickDraw GX does not fill the inner loop even when you specify the winding shape fill. Unlike the even-odd shape fill, which never fills an inner loop, winding shape fill considers contour direction when filling a shape:

- n If the inner loop and the outer loop have the same contour direction, winding shape fill causes QuickDraw GX to fill the inner loop as well as the outer loop, as shown in Figure 2-32.
- n If the inner loop and the outer loop have opposite contour directions, winding shape fill causes QuickDraw GX to fill the outer loop, but not the inner loop. The next section gives an example using path shapes.

For more information about contour direction and shape-filling algorithms, see “Shape Fill” beginning on page 2-12.

For more information about polygon shapes, see “Polygon Shapes” on page 2-22 and “Polygon Structures” on page 2-106.

For information about the functions you can use to create and draw polygons, see the description of the `GXNewPolygons` function on page 2-116 and the `GXDrawPolygons` function on page 2-161.

## Creating and Drawing Paths

---

Like a polygon contour, a path contour is a series of connected points. However, whereas a polygon contour is made up of straight lines, a path contour can contain both straight lines and curves. Therefore, the geometric points that make up a path contour can be on-curve points or off-curve control points. QuickDraw GX defines the `gxPath` structure to encapsulate a path contour geometry:

```
struct gxPath {
    long          vectors;
    long          controlBits[gxAnyNumber];
    struct gxPoint vector[gxAnyNumber];
};
```

## Geometric Shapes

The `vectors` field indicates the number of geometric points in the path and the `vector` array contains the geometric points themselves. The `controlBits` array specifies which geometric points are on-curve points and which are off-curve control points. A value of 0 indicates an on-curve point and a value of 1 indicates an off-curve point. For example, a `controlBits` field with the value

```
0x55555555 /* 0101 0101 0101 0101 ... */
```

indicates that every other point is an off-curve control point; the first point is on curve, the second point is off, and so on. As another example, a `controlBits` field value of

```
0x00000000 /* 0000 0000 0000 0000 ... */
```

indicates all points are on curve, which effectively creates a polygon.

Notice that the `controlBits` array allows you to specify sequential off-curve control points. For example, a `controlBits` value of

```
0xFFFFFFFF /* 1111 1111 1111 1111 ... */
```

indicates that all points are off curve. When you indicate that two control points in a row are off curve, QuickDraw GX assumes an on-curve point midway between them. (The example in Listing 2-17 on page 2-59 gives an example.)

Like the polygon shape, the path shape allows you to group any number of contours within a single QuickDraw GX shape. The `gxPaths` structure encapsulates the multiple-path geometry:

```
struct gxPaths {
    long          contours;
    struct gxPath contour[gxAnyNumber];
};
```

The `contours` field indicates the total number of contours (in other words, the total number of separate paths), and the `contour` array contains the path geometries.

### Creating Paths With a Single Contour

---

Since a `gxPaths` structure is of variable length and every element in it is of type `long`, you can define a path geometry as an array of `long` values. The sample function in Listing 2-16 shows how to define a path geometry as an array of `long` values, and then draw a path shape using the `GXDrawPaths` function. Since the `GXDrawPaths` function expects its first parameter to be a pointer to a `gxPaths` structure, the sample function casts the array of `long` values to the appropriate type using the expression

```
(gxPaths *) aPathGeometry
```

before sending the information to the `GXDrawPaths` function.

---

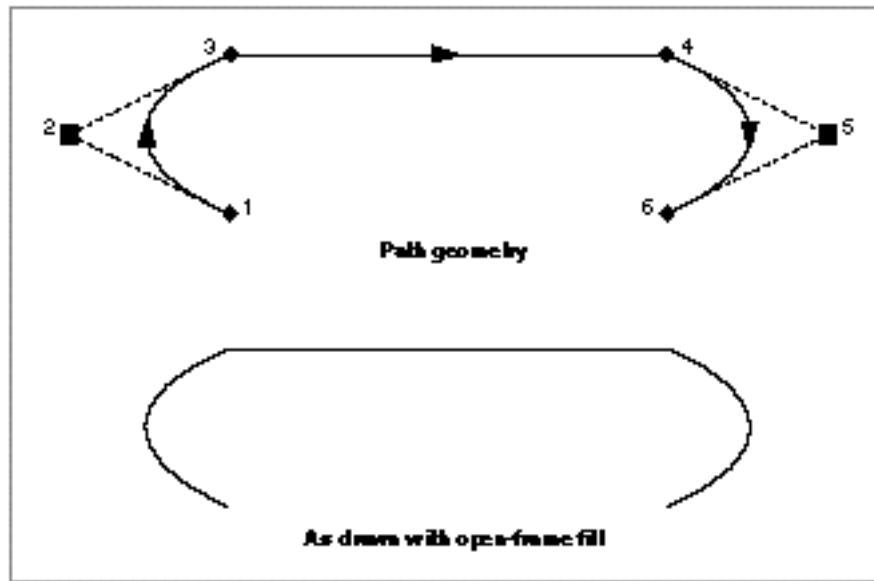
**Listing 2-16** Drawing a path shape

```
void DrawAPathShape(void)
{
    static long aPathGeometry[] = {1, /* number of contours */
                                    6, /* number of points */
                                    0x48000000, /* 0100 1000 */
                                    ff(50), ff(100), /* on */
                                    ff(0), ff(75), /* off */
                                    ff(50), ff(50), /* on */
                                    ff(150), ff(50), /* on */
                                    ff(200), ff(75), /* off */
                                    ff(150), ff(100)}; /* on */

    GXDrawPaths((gxPaths *) aPathGeometry, gxOpenFrameFill);
}
```

The path defined in this example has four on-curve points and two off-curve points. When drawn with the open-frame shape fill, it contains two curves and one straight line, as shown in Figure 2-33.

**Figure 2-33** A path



The sample function from Listing 2-16 draws the path without creating a path shape. It could instead create a path shape with the `GXNewPaths` function:

```
aPathsShape = GXNewPaths((gxPaths *) aPathGeometry);
```

and then draw it using the `GXDrawShape` function:

```
GXDrawShape(aPathsShape);
```

You can also create path shapes using the `GXNewShape` function:

```
aPathsShape = GXNewShape(gxPathType);
GXSetPaths(aPathsShape, (gxPaths *) aPathGeometry);
```

or by using the `GXNewShapeVector` function:

```
aPathsShape = GXNewShapeVector(gxPathType, aPathGeometry);
```

Notice that in this case you do not have to cast the `aPathGeometry` array to be a pointer to a `gxPaths` structure. The `GXNewShapeVector` function expects an array of long values.

Although the `GXDrawPaths` function (shown in Listing 2-16) allows you to specify a shape fill, the `GXDrawShape` function does not. If you create a path shape and you want it to have a different shape fill than the default path shape, you must indicate the desired shape fill using the `GXSetShapeFill` function—for example,

```
GXSetShapeFill(aPathsShape, gxInverseEvenOddFill);
```

For more information about shape fills, see “Shape Fill” beginning on page 2-12.

### Creating Paths Using Only Off-Curve Points

---

The sample function in Listing 2-17 shows how you can create a path using only off-curve control points. The path defined in this example contains four control points, and the `controlBits` field is set to

```
0xF0000000 /* 1111 0000 0000 0000 0000 ... */
```

which indicates that the first four points are off curve. The path contains only four points, and therefore they are all off curve.

---

**Listing 2-17** Creating a path using only off-curve control points

```
void CreateRoundPath(void)
{
    gxShape      aPathShape;

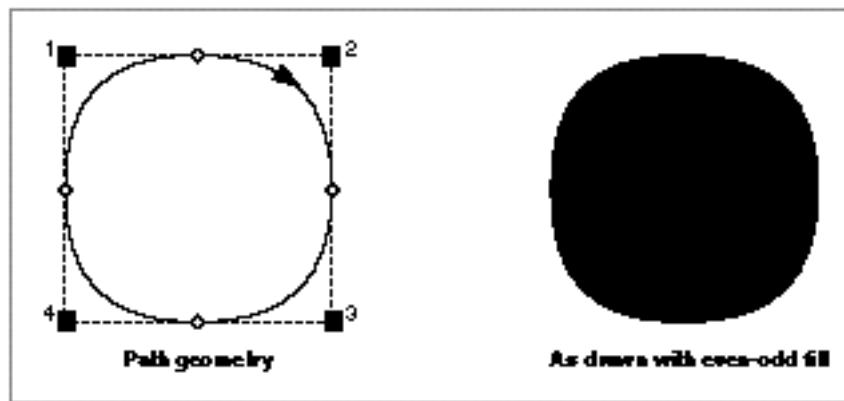
    static long aPathGeometry[] = {1, /* number of contours */
                                    4, /* number of points */
                                    0xF0000000, /* 1111 0000 ... */
                                    ff(50), ff(50), /* off */
                                    ff(150), ff(50), /* off */
                                    ff(150), ff(150), /* off */
                                    ff(50), ff(150)}; /* off */

    aPathShape = GXNewPaths((gxPaths *) aPathGeometry);

    GXDrawShape(aPathShape);
}
```

The four off-curve control points in this example form a square; the path that they define is a rounded square, as shown in Figure 2-34.

**Figure 2-34** A round path shape



Notice that the path is filled with the even-odd shape fill, which is the default for path shapes. You could, however, specify any shape fill for this path except the open-frame shape fill. The open-frame shape fill requires that the first and last points of the contour be on-curve points, and this path has no on-curve points.

### Creating Paths With Multiple Contours

The sample function in Listing 2-18 shows how a single path shape can contain more than one path contour. The path shape defined in this example includes the round path from the previous example as well as a second round path, entirely contained within the first.



**Listing 2-18** Creating a path with concentric contours

---

```

void CreateHollowCircles(void)
{
    gxShape  aPathShape;

    static long aPathGeometry[] = {2, /* number of contours */
                                    4, /* number of points */
                                    0xF0000000, /* 1111 0000 ... */
                                    ff(50), ff(50), /* off */
                                    ff(150), ff(50), /* off */
                                    ff(150), ff(150), /* off */
                                    ff(50), ff(150), /* off */

                                    4, /* number of points */
                                    0xF0000000, /* 1111 0000 ... */
                                    ff(65), ff(65), /* off */
                                    ff(135), ff(65), /* off */
                                    ff(135), ff(135), /* off */
                                    ff(65), ff(135)}; /* off */

    aPathShape = GXNewPaths((gxPaths *) aPathGeometry);
    GXSetShapeFill(aPathShape, gxClosedFrameFill);

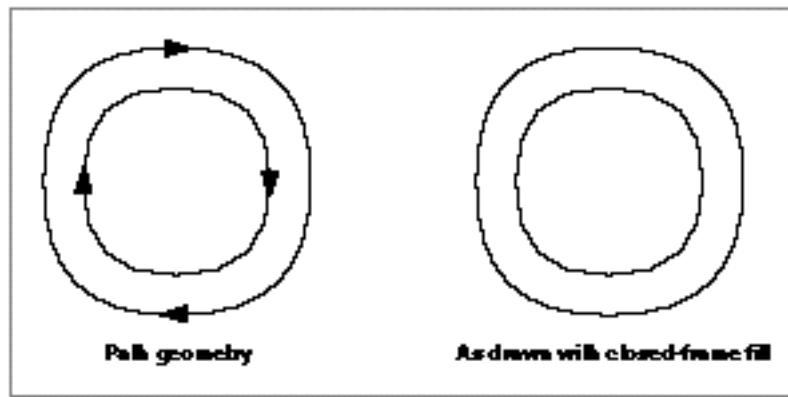
    GXDrawShape(aPathShape);

    GXDisposeShape(aPathShape);
}

```

The result of this function is shown in Figure 2-35.

**Figure 2-35** A path shape with two concentric clockwise contours and closed-frame shape fill

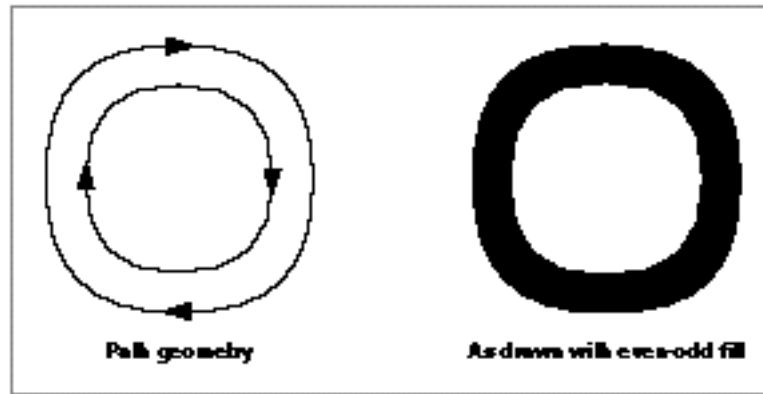


You can change the shape fill of this polygon by removing this line of code from the sample function in Listing 2-18:

```
GXSetShapeFill(aPolygonsShape, gxClosedFrameFill);
```

If you don't specify a shape fill, the `GXNewPaths` function uses the shape fill from the default path shape, which is the even-odd shape fill (unless you change it using the `GXGetDefaultShape` and `GXSetShapeFill` functions). The path shape resulting from an even-odd shape fill is shown in Figure 2-36.

**Figure 2-36** A path shape with two concentric clockwise contours and even-odd shape fill

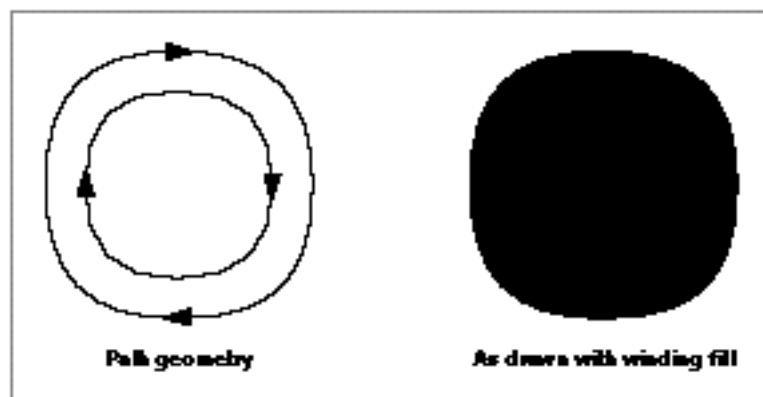


Notice that the even-odd shape fill causes QuickDraw GX to fill in the outer contour, but not the inner contour. However, if you specify the winding shape fill for this path using the call

```
GXSetShapeFill(aPathShape, gxWindingFill);
```

the resulting shape would appear as shown in Figure 2-37.

**Figure 2-37** A path shape with two concentric clockwise contours and winding shape fill



Unlike the even-odd shape fill, the winding shape fill causes QuickDraw GX to fill inner contours—as long as the inner contour has the same contour direction as the outer contour. If the inner contour and the outer contour have opposite contour directions, neither the even-odd shape fill nor the winding shape fill will fill the inner contour.

For example, if you change the direction of the inner contour from the previous example by reversing the order of the second path's geometric points, as in the declaration

```
static long aPathGeometry[] = {2, /* number of contours */
                                4, /* number of points */
                                0xF0000000, /* 1111 0000 */
                                ff(50), ff(50), /* off */
                                ff(150), ff(50), /* off */
                                ff(150), ff(150), /* off */
                                ff(50), ff(150), /* off */

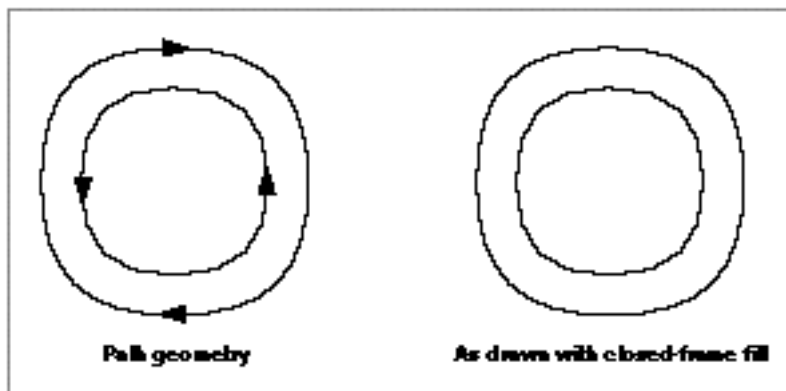
                                4, /* number of points */
                                0xF0000000, /* 1111 0000 */
                                ff(65), ff(135), /* off */
                                ff(135), ff(135), /* off */
                                ff(135), ff(65), /* off */
                                ff(65), ff(65)}; /* off */
```

and set the shape fill to the closed-frame shape fill using the call

```
GXSetShapeFill(aPathShape, gxClosedFrameFill);
```

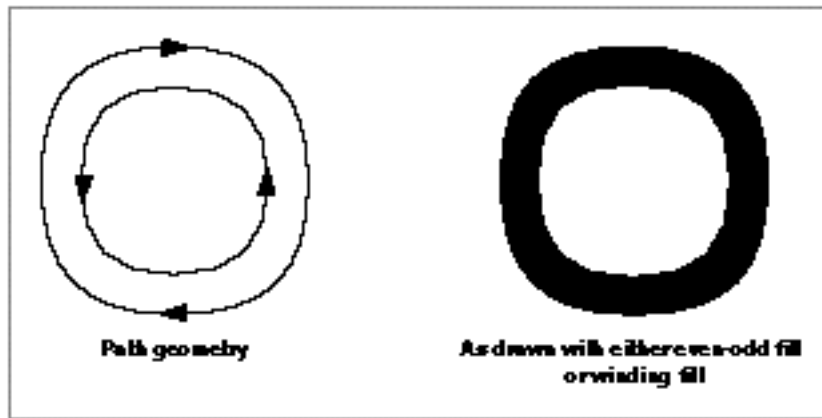
the resulting shape has contours with opposite contour directions, as depicted in Figure 2-38.

**Figure 2-38** A path shape with an internal counterclockwise contour and closed-frame shape fill



Since the outer contour and the inner contour have opposite contour directions, neither the even-odd shape fill nor the winding shape fill cause QuickDraw GX to fill the inner contour, as shown in Figure 2-39.

**Figure 2-39** A path shape with even-odd or winding shape fill



For more information about contour direction and shape-filling algorithms, see “Shape Fill” on page 2-12.

For more information about path shapes, see “Path Shapes” on page 2-25 and “Path Structures” on page 2-107.

For information about the functions you can use to create and draw paths, see the description of the `GXNewPaths` function on page 2-117 and the `GXDrawPaths` function on page 2-162.

## Converting Between Geometric Shape Types

QuickDraw GX provides the `GXGetShapeType` and `GXSetShapeType` functions to allow you to manipulate a shape’s type. The `GXGetShapeType` function simply returns the value of the shape type property for a specified shape. For geometric shapes, the possible values returned from this function are

```
n gxEmptyType
n gxFullType
n gxPointType
n gxLineType
n gxCurveType
n gxRectangleType
```

## Geometric Shapes

n `gxPolygonType`

n `gxPathType`

The `GXSetShapeType` function allows you to change the shape type of an existing shape. In doing so, this function often has to reinterpret the geometry of the shape. This reinterpretation is called **type conversion**. Sometimes the conversion makes sense and doesn't lose any data. For example, you might want to convert a line shape to a polygon shape so that you can add more contours to the shape. Some conversions, however, aren't as useful and data can be lost. For example, converting a complex path shape to a point shape can result in the loss of a significant amount of data.

In general, when converting between geometric shape types, QuickDraw GX exhibits different behavior in these four situations:

- n when converting other geometric shapes to an empty shape or a full shape
- n when converting other geometric shapes to a point, line, or rectangle
- n when converting other geometric shapes to a curve
- n when converting other geometric shapes to a polygon or path

When converting to an empty shape or a full shape, all the information in the original shape's geometry is lost—the result is simply an empty shape or a full shape, respectively.

The following three subsections discuss the other cases in more detail.

### Converting Shapes to Points, Lines, and Rectangles

---

When converting a shape to a point, line, or rectangle, QuickDraw GX uses the bounding rectangle of the original shape. (Bounding rectangles are defined in the chapter “Geometric Operations” in this book. For an example, see Figure 2-41 on page 2-68.) QuickDraw GX uses the bounding rectangle differently, depending on which shape type you are converting to:

- n When you convert to a rectangle shape, the resulting rectangle is the bounding rectangle of the original shape.
- n When you convert to a line shape, the result is a line that runs from the upper-left corner of the original shape's bounding rectangle to the lower-right corner.
- n When you convert to a point, the resulting point is the point at the upper-left corner of the bounding rectangle of the original shape.

Listing 2-19 creates a path shape, which is converted subsequently to a rectangle shape, then to a line shape, and finally to a point shape.

**Listing 2-19** Creating a figure-eight path shape

---

```
void CreateFigureEight(void)
{
    gxShape      aPathShape;

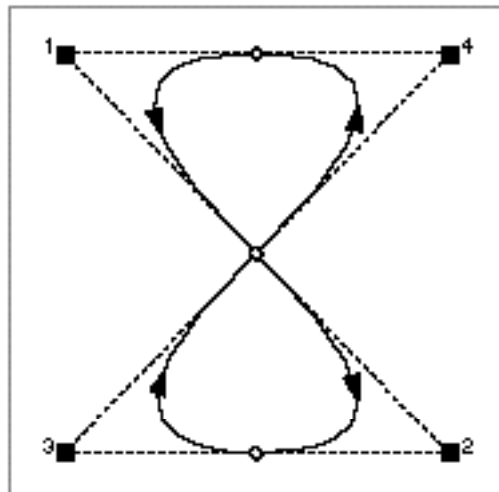
    static long figureEightGeometry[] = {1, /* number of contours */
                                          4, /* number of points */
                                          0xF0000000, /* 1111 ... */
                                          ff(50), ff(50), /* off */
                                          ff(200), ff(200), /* off */
                                          ff(50), ff(200), /* off */
                                          ff(200), ff(50)}; /* off */

    aPathShape = GXNewPaths((gxPaths *) figureEightGeometry);
    GXSetShapeFill(aPathShape, gxClosedFrameFill);

    GXDrawShape(aPathShape);
}
```

The resulting path geometry is shown in Figure 2-40.

**Figure 2-40** A figure-eight path shape



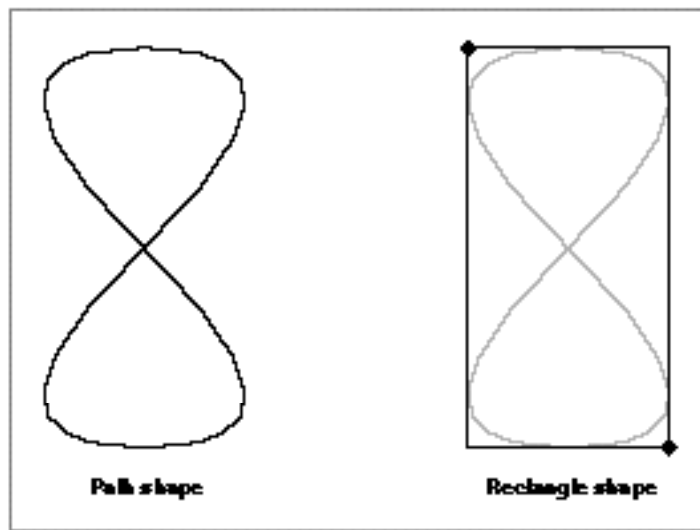
## Geometric Shapes

If you convert this shape to a rectangle shape, using the call

```
GXSetShapeType(aPathShape, gxRectangleType);
```

the resulting shape is the bounding rectangle for the original path shape, as shown in Figure 2-41.

**Figure 2-41** A path shape before and after conversion to a rectangle shape





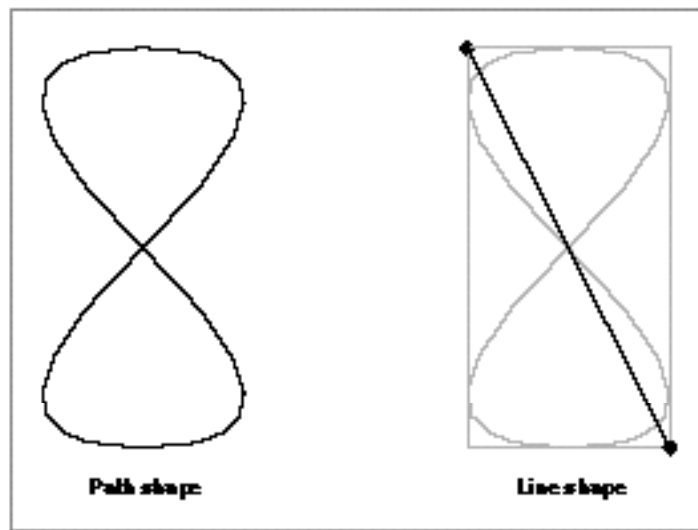
## Geometric Shapes

If you convert the original path shape to a line shape, using the call

```
GXSetShapeType(aPathShape, gxLineType);
```

the resulting shape is a diagonal line from the upper-left corner of the path's bounding rectangle to its lower-right corner, as shown in Figure 2-42.

**Figure 2-42** A path shape before and after conversion to a line shape



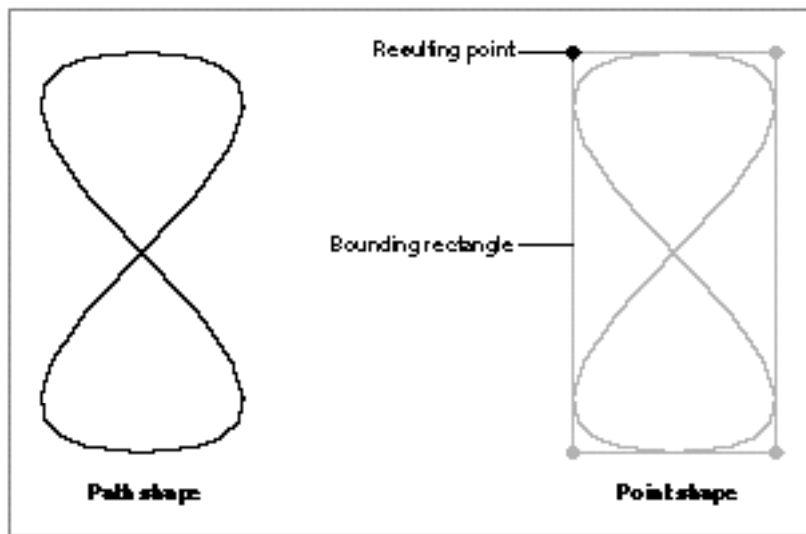
## Geometric Shapes

Finally, if you convert the original path shape to a point shape, using the call

```
GXSetShapeType(aPathShape, gxPointType);
```

the resulting shape is the point at the upper-left corner of the path's bounding rectangle, as shown in Figure 2-43.

**Figure 2-43** A path shape before and after conversion to a point shape



The next two sections give examples of converting to the curve shape type and of converting to the polygon and path shape types.

For more information about the `GXSetShapeType` function, see the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects*.

## Converting Shapes to Curve Shapes

---

When converting other geometric shapes to a curve shape, QuickDraw GX takes one of these approaches:

- n When converting a point to a curve, QuickDraw GX sets each of the three geometric points of the curve to be the same as the original point.
- n When converting a line to a curve, QuickDraw GX sets the first point and last point of the curve to be the same as the first point and last point of the line and sets the off-curve control point to be same as the last point of the line, which results in the curve being a straight line.
- n When converting a rectangle to a curve, QuickDraw GX sets the first point of the curve to be the upper-left corner of the rectangle and the last point of the curve to be the lower-right corner of the rectangle. The off-curve control point is set to be the same as the last point, which results in the curve being a straight line.
- n When converting a polygon or a path to a curve, QuickDraw GX sets the three geometric points of the curve to be the first three geometric points of the original shape.

The sample function in Listing 2-20 creates a line shape and converts it to a curve.

---

**Listing 2-20** Converting a line to a curve

```
void ConvertLineToCurve(void)
{
    gxShape      aLineShape;

    static gxLine diagonalGeometry = {ff(50), ff(50),
                                      ff(150), ff(150)};

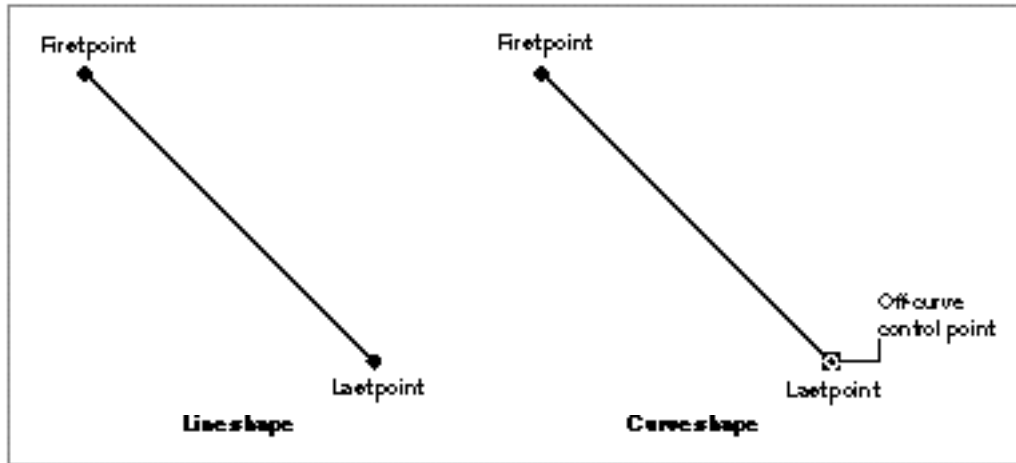
    aLineShape = GXNewLine(&diagonalGeometry);
    GXSetShapeType(aLineShape, gxCurveType);

    GXDrawShape(aLineShape);

    GXDisposeShape(aLineShape);
}
```

The original line shape and the resulting curve shape are shown in Figure 2-44.

**Figure 2-44** A line shape before and after conversion to a curve shape



Notice that the converted curve looks just like the original line. The only difference between the two shapes is that the curve shape has an additional off-curve control point, which is set to be identical to the last point.

The sample function in Listing 2-21 creates a rectangle shape and converts it to a curve shape.

**Listing 2-21** Converting a rectangle to a curve

```
void ConvertRectangleToCurve(void)
{
    gxShape aRectangleShape;

    static gxRectangle rectangleGeometry = {ff(50), ff(50),
                                             ff(150), ff(150)};

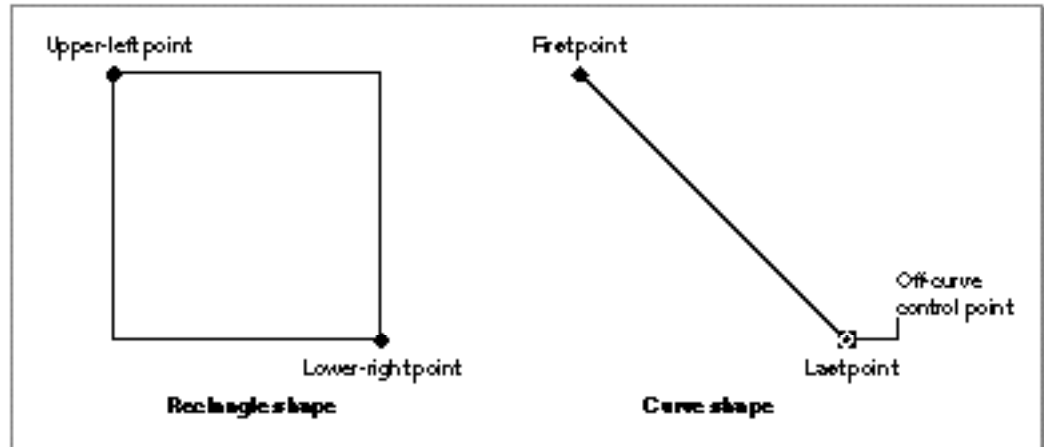
    aRectangleShape = GXNewRectangle(&rectangleGeometry);
    GXSetShapeType(aRectangleShape, gxCurveType);

    GXDrawShape(aRectangleShape);

    GXDisposeShape(aLineShape);
}
```

The original rectangle and the resulting curve are both shown in Figure 2-45.

**Figure 2-45** A rectangle shape before and after conversion to a curve shape



As in the previous example, the off-curve control point of the curve shape is set to be the same as the last point, which results in the curve shape being a straight line.

The next example, shown in Listing 2-22, shows how QuickDraw GX converts a polygon shape to a curve shape.

**Listing 2-22** Converting a polygon shape to a curve shape

```
void ConvertPolygonToCurve(void)
{
    gxShape aPolygonShape;

    static long aPolygonGeometry[] = {1, /* number of contours */
                                      4, /* number of points */
                                      ff(50), ff(50),
                                      ff(150), ff(50),
                                      ff(150), ff(150),
                                      ff(50), ff(150)};

    aPolygonShape = GXNewPolygons((gxPolygons *) aPolygonGeometry);
    GXSetShapeType(aPolygonShape, gxCurveType);
}
```

## Geometric Shapes

```

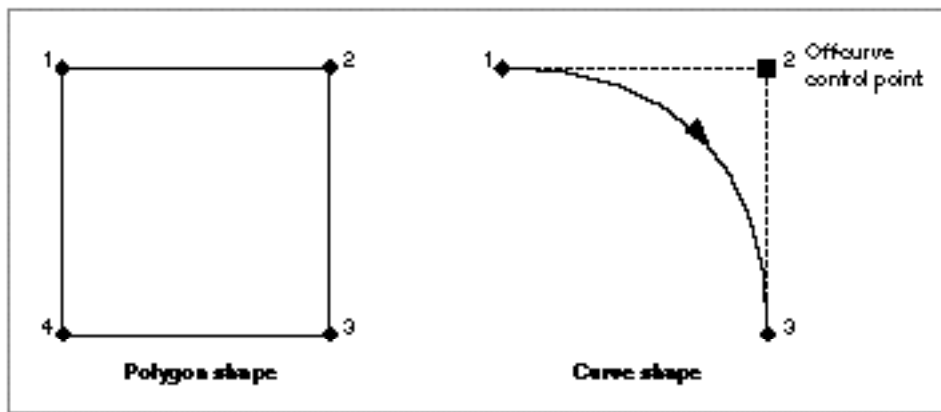
GXDrawShape(aPolygonShape);

GXDisposeShape(aPolygonShape);
}

```

In this example, QuickDraw GX sets the three geometric points of the resulting curve to be the first three geometric points of the original polygon. (Converting from path shapes to curve shapes works in the same way.) The original polygon and the resulting curve are shown in Figure 2-46.

**Figure 2-46** A polygon shape before and after conversion to a curve shape



Notice that even though the polygon in this example looks the same as the rectangle in Figure 2-45, the converted curve shape looks quite different.

The next section gives examples of converting shapes to polygon and path shapes.

For more information about the `GXSetShapeType` function in general, see the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects*.

### Converting Shapes to Polygons and Paths

When converting other geometric shapes to polygon or path shapes, the original shapes don’t lose any geometric information. For example, when you convert a line shape to a path shape, the resulting path shape contains one contour with two geometric points, both on curve—an exact duplicate of the original line.

You can even convert a curve shape to a polygon shape without losing geometric information, although the result does draw differently. The resulting polygon has one contour and three geometric points—the same three geometric points as the original curve. If you convert the polygon back to a curve, you end up with the original curve.

When you convert a rectangle shape to a polygon shape, as shown in Listing 2-23, the original shape and the resulting shape look exactly the same.

**Listing 2-23** Converting a rectangle shape to a polygon shape

```
void ConvertRectangleToPolygon(void)
{
    gxShape aRectangleShape;

    static gxRectangle rectangleGeometry = {ff(50), ff(50),
                                            ff(150), ff(150)};

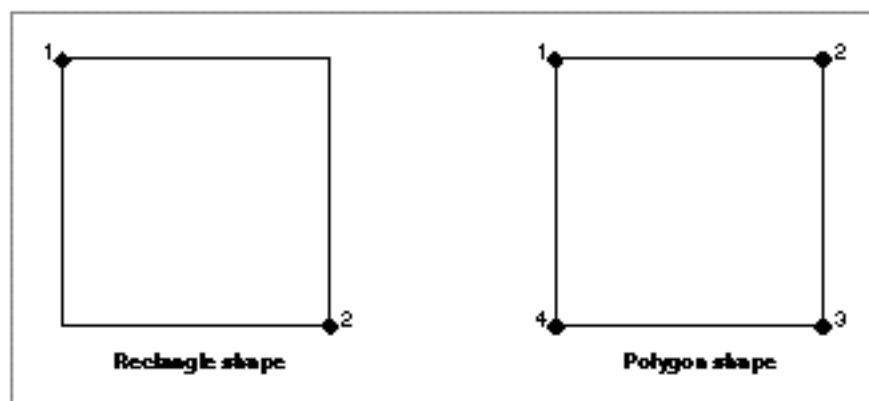
    aRectangleShape = GXNewRectangle(&rectangleGeometry);
    GXSetShapeType(aRectangleShape, gxPolygonType);

    GXDrawShape(aRectangleShape);

    GXDisposeShape(aRectangleShape);
}
```

The original rectangle and the resulting polygon are shown in Figure 2-47.

**Figure 2-47** A rectangle shape before and after conversion to a polygon shape



## Geometric Shapes

Converting from a path shape to a polygon shape, however, does lose geometric information. The resulting polygon contains the same geometric points as the original path; however, the points are all considered on-curve points. The original information about which points were on curve and which points were off curve is lost during this conversion.

As an example, Listing 2-24 creates a path shape and converts it to a polygon shape.

---

**Listing 2-24** Converting a path shape to a polygon shape

```
void ConvertPathToPolygon(void)
{
    gxShape  aPathShape;

    static long figureEightGeometry[] = {1, /* # of contours */
                                          4, /* # of points */
                                          0xF0000000, /* 1111 ... */
                                          ff(50), ff(50), /* off */
                                          ff(200),ff(200), /* off */
                                          ff(50), ff(200), /* off */
                                          ff(200),ff(50)}; /* off */

    aPathShape = GXNewPaths((gxPaths *) figureEightGeometry);
    GXSetShapeFill(aPathShape, gxHollowFill);
    GXSetShapeType(aPathShape, gxPolygonType);

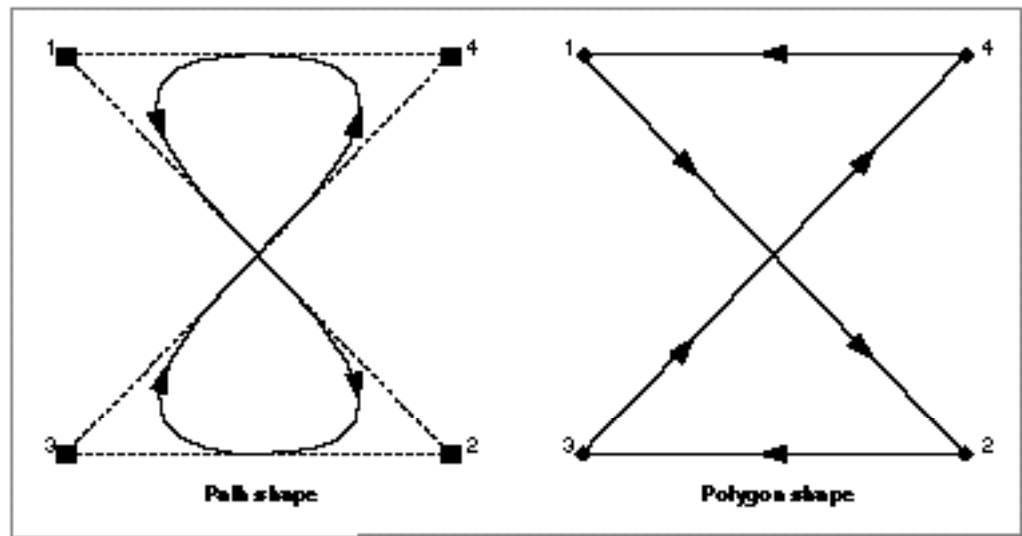
    GXDrawShape(aPathShape);

    GXDisposeShape(aPathShape);
}
```



Figure 2-48 shows the original path shape and the resulting polygon shape.

**Figure 2-48** A path shape before and after conversion to a polygon shape



**Note**

You can request that QuickDraw GX calculate a better polygon approximation to a path by setting the curve error property of the path shape's style object before calling the `GXSetShapeType` function. See the next chapter, "Geometric Styles," for examples. [u](#)

## Geometric Shapes

Converting in the other direction, however—from a polygon shape to a path shape—retains all of the geometry information and the resulting path shape looks exactly the same as the original polygon shape. The sample function in Listing 2-25 gives an example.

---

**Listing 2-25** Converting a polygon shape to a path shape

```
void ConvertPolygonToPath(void)
{
    gxShape      aPolygonShape;

    static long aPolygonGeometry[] = {2, /* number of contours */
                                       3, /* number of points */
                                       ff(50), ff(50),
                                       ff(100), ff(80),
                                       ff(50), ff(110),
                                       3, /* number of points */
                                       ff(200), ff(50),
                                       ff(150), ff(80),
                                       ff(200), ff(110)};

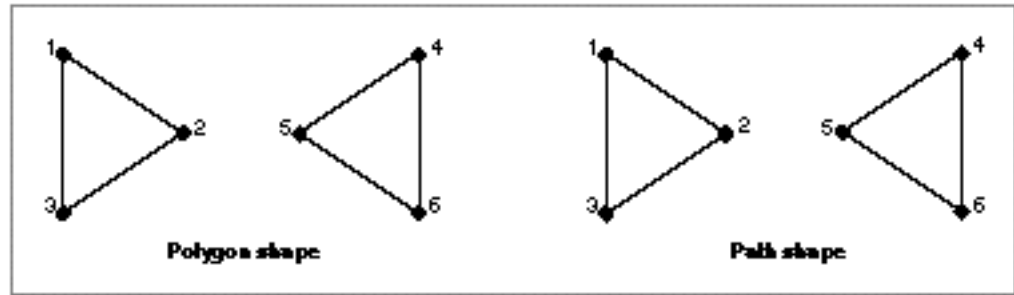
    aPolygonShape = GXNewPolygons((gxPolygons *)
                                   aPolygonGeometry);
    GXSetShapeType(aPolygonShape, gxPathType);

    GXDrawShape(aPolygonShape);

    GXDisposeShape(aPolygonShape);
}
```

The original polygon shape and the converted path shape are shown in Figure 2-49.

**Figure 2-49** Polygon shape with two contours before and after conversion to a path shape



For more information about the `GXSetShapeType` function, see the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects*.

## Replacing Geometric Points

The `GXSetPoint`, `GXSetLine`, `GXSetCurve`, `GXSetRectangle`, `GXSetPolygons`, and `GXSetPaths` functions allow you to replace the geometry of an existing shape. The limitation of these functions is that you must replace the entire geometry at once.

QuickDraw GX provides other, more sophisticated, mechanisms for editing shape geometries. The `GXSetShapePoints` function, which is illustrated in this section, allows you to replace individual geometric points within a shape’s geometry. The `GXSetPolygonParts`, `GXSetPathParts`, and `GXSetShapeParts` functions, which are discussed in the next three sections, provide even more ways to edit the geometries of shapes.

## Geometric Shapes

The sample function in Listing 2-26 creates a path shape and uses the `GXSetShapePoints` function to replace two of the path's geometric points.

---

**Listing 2-26** Replacing geometric points

```
void ReplaceTopTwoControlPoints(void)
{
    gxShape  aPathShape;

    static long twoCurveGeometry[] = {1, /* number of contours */
                                       6, /* number of points */
                                       0x48000000, /* 0100 1000 */
                                       ff(100), ff(150), /* on */
                                       ff(50),  ff(100), /* off */
                                       ff(100), ff(50), /* on */
                                       ff(200), ff(50), /* on */
                                       ff(250), ff(100), /* off */
                                       ff(200), ff(150)}; /* on */

    static gxPoint newTopGeometry[] = {ff(140), ff(50),
                                       ff(160), ff(50)};

    aPathShape = GXNewPaths((gxPaths *) twoCurveGeometry);
    GXSetShapeFill(aPathShape, gxOpenFrameFill);

    GXSetShapePoints(aPathShape, 3, 2, newTopGeometry);

    GXDrawShape(aPathShape);

    GXDisposeShape(aPathShape);
}
```

The `GXSetShapePoints` function takes four parameters:

- n a reference to the shape to edit
- n the index of the first geometric point to be replaced
- n the number of geometric points to replace
- n an array containing the new geometric points

Therefore, the line of code from the sample function in Listing 2-26

```
GXSetShapePoints(aPathsShape, 3, 2, newTopGeometry);
```

replaces the third and fourth geometric point from the original path shape with the two geometric points in the `newTopGeometry` array.

Figure 2-50 shows the path shape before the geometric points are replaced.

**Figure 2-50** A path shape with a flat top

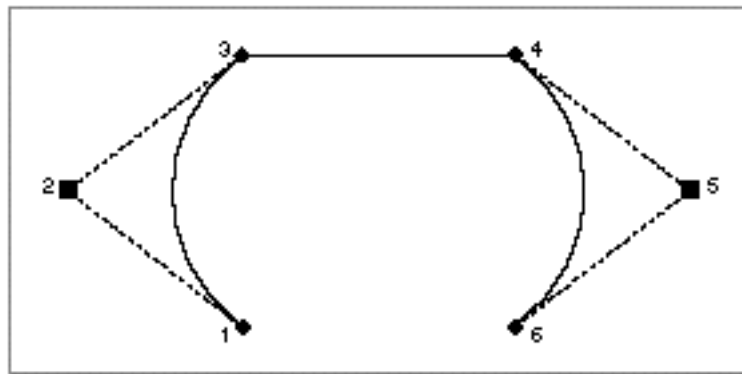
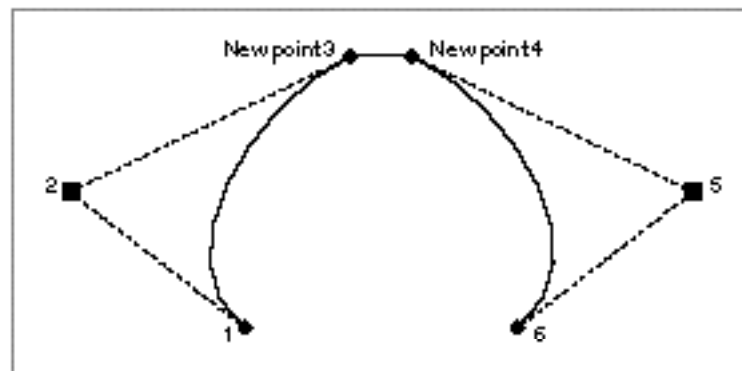


Figure 2-51 shows the path shape after the geometric points are replaced.

**Figure 2-51** A path shape with geometric points replaced



For more information about the `GXSetShapePoints` function, see page 2-142.

The next three sections give examples of functions that allow you even more control in editing the geometric points of a shape's geometry.

## Editing Polygon Parts

---

QuickDraw GX provides six functions that allow sophisticated editing of geometric shapes:

- n The `GXGetPolygonParts` and `GXSetPolygonParts` functions allow you to extract information from a polygon geometry, replace information in the geometry, remove information in the geometry, and insert new information in the geometry.
- n The `GXGetPathParts` and `GXSetPathParts` functions allow you to extract, replace, remove, and insert information in a path shape's geometry.
- n The `GXGetShapeParts` and `GXSetShapeParts` functions allow you to extract, replace, remove, and insert information in any shape's geometry.

This section gives examples of the `GXGetPolygonParts` and `GXSetPolygonParts` functions. The next two sections show how to edit path shape geometries and shape geometries in general.

Listing 2-27 creates a polygon shape with two contours. Later examples in this section use this polygon shape to demonstrate editing polygon parts.

---

**Listing 2-27**    Creating a polygon shape with two contours

```
void CreateTwoAngles(void)
{
    gxShape      aPolygonShape;

    static long twoAngleGeometry[] = {2, /* number of contours */
                                       3, /* number of points */
                                       ff(100), ff(150),
                                       ff(50), ff(100),
                                       ff(100), ff(50),
                                       3, /* number of points */
                                       ff(200), ff(50),
                                       ff(250), ff(100),
                                       ff(200), ff(150)};

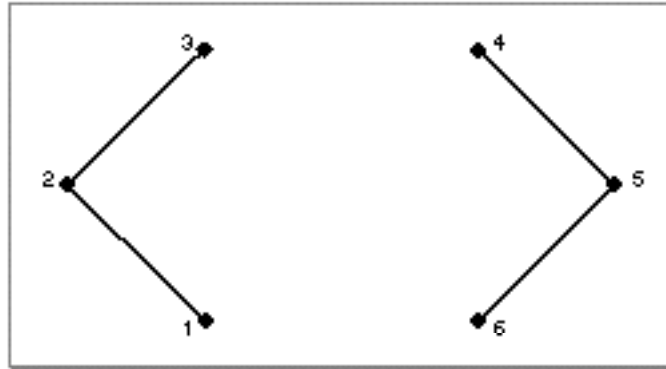
    aPolygonShape = GXNewPolygons((gxPolygons *)
                                   twoAngleGeometry);
    GXSetShapeFill(aPolygonShape, gxOpenFrameFill);

    GXDrawShape(aPolygonShape);

    GXDisposeShape(aPolygonShape);
}
```

The result of this sample function is shown in Figure 2-52.

**Figure 2-52** A polygon shape with two contours



The `GXGetPolygonParts` function allows you to extract geometric points from the geometry of an existing polygon shape and put them into a new polygon geometry. This function takes four parameters:

- n a reference to the existing polygon shape
- n the index of the first desired geometric point
- n the number of geometric points to include
- n a pointer to a polygon geometry in which to store the extracted geometric points

The `GXGetPolygonParts` function returns as its function result the number of bytes necessary to contain the extracted polygon. Therefore, you typically call `GXGetPolygonParts` twice—once to determine the size of extracted polygon and once to extract the polygon. For example, if you declare the variable

```
long    byteCount;
```

you could determine the number of bytes necessary to extract the top half of the polygon geometry in Figure 2-52 using this line of code:

```
byteCount = GXGetPolygonParts(aPolygonsShape, 2, 4, nil);
```

In this example, setting the final parameter to `nil` indicates that you want to determine the number of bytes necessary to hold the extracted polygon geometry, but you do not want to actually extract the polygon geometry. The values of 2 and 4 for the second and third parameters indicate that the `GXGetPolygonParts` function should determine the number of bytes necessary to hold an extracted polygon geometry that contains geometric points 2, 3, 4, and 5 from the polygon geometry in Figure 2-52.

## Geometric Shapes

You can then use this byte count to allocate enough memory to hold the extracted polygon geometry:

```
gxPolygons *topHalfGeometry;
topHalfGeometry = (gxPolygons *) NewPtr(byteCount);
```

Finally, you can extract the polygon geometry by calling `GXGetPolygonParts` again:

```
GXGetPolygonParts(aPolygonsShape, 2, 4, topHalfGeometry);
```

The sample function in Listing 2-28 creates the polygon shape from the previous example, extracts the second through the fifth geometric points and puts them into a separate geometry, and then replaces the geometry of the original polygon shape with the extracted geometry.

---

**Listing 2-28** Extracting part of a polygon shape

```
void ExtractTopPartOfPolygon(void)
{
    gxShape      aPolygonShape;

    static long twoAngleGeometry[] = {2, /* number of contours */
                                       3, /* number of points */
                                       ff(100), ff(150),
                                       ff(50), ff(100),
                                       ff(100), ff(50),
                                       3, /* number of points */
                                       ff(200), ff(50),
                                       ff(250), ff(100),
                                       ff(200), ff(150)};

    long        byteCount;

    gxPolygons *topHalfGeometry;

    aPolygonShape = GXNewPolygons((gxPolygons *)
                                   twoAngleGeometry);
    GXSetShapeFill(aPolygonShape, gxOpenFrameFill);
```



## Geometric Shapes

```

byteCount = GXGetPolygonParts(aPolygonShape, 2, 4, nil);
topHalfGeometry = (gxPolygons *) NewPtr(byteCount);
GXGetPolygonParts(aPolygonShape, 2, 4, topHalfGeometry);

GXSetPolygons(aPolygonShape, topHalfGeometry);

GXDrawShape(aPolygonShape);

GXDisposeShape(aPolygonShape);
}

```

The resulting polygon shape is shown in Figure 2-53.

---

**Figure 2-53** A polygon shape extracted from a larger polygon shape



Compare this polygon shape with the polygon shape shown in Figure 2-52.

Like the `GXSetShapePoints` function discussed in the previous section, the `GXSetPolygonParts` function allows you to replace geometric points within a polygon shape's geometry. However, the `GXSetPolygonParts` function allows you even more editing control. With it, you can also remove geometric points, insert geometric points, and break a polygon shape into multiple contours.

## Geometric Shapes

As an example of replacing geometric points in a polygon geometry, the sample function in Listing 2-29 creates two polygon geometries: the two-angle polygon geometry from Figure 2-52 and another polygon geometry consisting of a single point. The sample function creates the two-angle polygon shape as in Listing 2-27 and then replaces its third and fourth geometric points with the single geometric point of the other polygon geometry.

---

**Listing 2-29** Replacing geometric points of a polygon shape

```
void ReplaceControlPoints(void)
{
    gxShape      aPolygonShape;

    static long twoAngleGeometry[] = {2, /* number of contours */
                                       3, /* number of points */
                                       ff(100), ff(150),
                                       ff(50), ff(100),
                                       ff(100), ff(50),
                                       3, /* number of points */
                                       ff(200), ff(50),
                                       ff(250), ff(100),
                                       ff(200), ff(150)};

    static long newTopGeometry[] = {1, /* number of contours */
                                     1, /* number of points */
                                     ff(150), ff(50)};

    aPolygonShape = GXNewPolygons((gxPolygons *)
                                   twoAngleGeometry);
    GXSetShapeFill(aPolygonShape, gxOpenFrameFill);

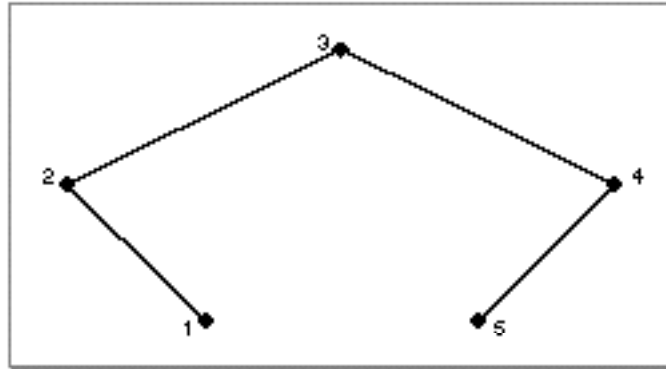
    GXSetPolygonParts(aPolygonShape, 3, 2,
                     (gxPolygons *) newTopGeometry,
                     gxBreakNeitherEdit);

    GXDrawShape(aPolygonShape);

    GXDisposeShape(aPolygonShape);
}
```

Figure 2-54 shows the result of the sample function in Listing 2-29.

**Figure 2-54** A polygon with two geometric points replaced by a single geometric point

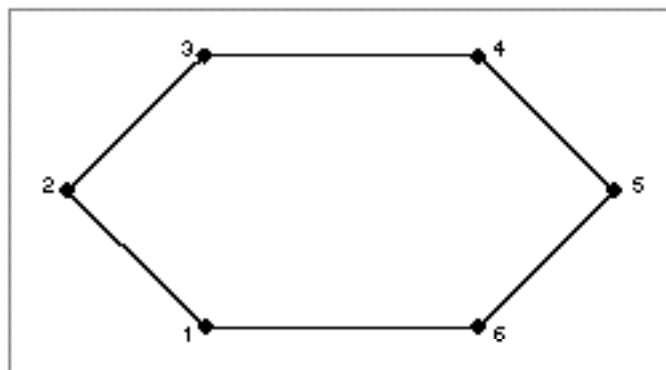


Notice that, whereas the `GXSetShapePoints` function limited you to replacing geometric points on a point-by-point basis, the `GXSetPolygonParts` function allows you to replace any number of geometric points in the original geometry with any number of new geometric points contained in an arbitrary polygon geometry.

Since the `GXSetPolygonParts` function allows you to insert an arbitrary polygon geometry into the geometry of an existing polygon shape, you can use this function to break a single polygon contour into multiple contours. In fact, the final parameter to `GXSetPolygonParts` allows you to specify how the resulting polygons shape should be broken up. In the previous example, the `gxBreakNeitherEdit` constant indicated that the resulting polygon should not be broken into separate contours.

The next example, shown in Listing 2-30, first creates a polygon shape similar to the two-angle polygons shape, except in this example the two contours are connected, as shown in Figure 2-55.

**Figure 2-55** A polygon shape with one contour



## Geometric Shapes

The sample function then uses the `GXSetPolygonParts` function to insert a new geometric point in the center of the polygon.

---

**Listing 2-30** Inserting a geometric point in a polygon shape

```
void CreateHollowPolygon(void)
{
    gxShape      aPolygonShape;

    static long twoAngleGeometry[] = {1, /* number of contours */
                                       6, /* number of points */
                                       ff(100), ff(150),
                                       ff(50), ff(100),
                                       ff(100), ff(50),
                                       ff(200), ff(50),
                                       ff(250), ff(100),
                                       ff(200), ff(150)};

    static long newCenterGeometry[] = {1, /* number of contours */
                                       1 /* number of points */ ,
                                       ff(150), ff(100)};

    aPolygonShape = GXNewPolygons((gxPolygons *)
                                  twoAngleGeometry);
    GXSetShapeFill(aPolygonShape, gxClosedFrameFill);

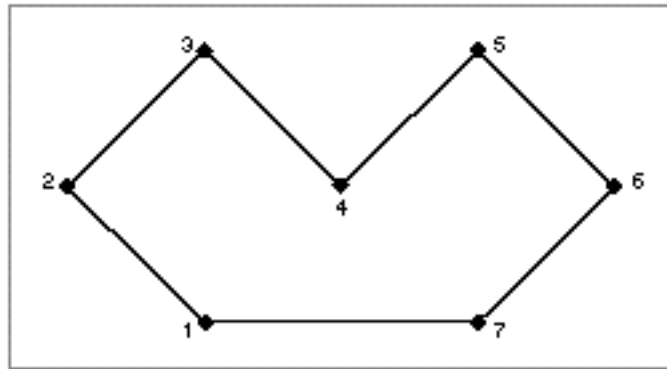
    GXSetPolygonParts(aPolygonShape, 4, 0,
                     (gxPolygons *) newCenterGeometry,
                     gxBreakNeitherEdit);

    GXDrawShape(aPolygonShape);

    GXDisposeShape(aPolygonShape);
}
```

Since this sample function specifies the `gxBreakNeitherEdit` constant as the final parameter to the `GXSetPolygonParts` function, the resulting polygon has a single contour, as shown in Figure 2-56.

**Figure 2-56** A polygon shape edited with the `gxBreakNeitherEdit` flag set

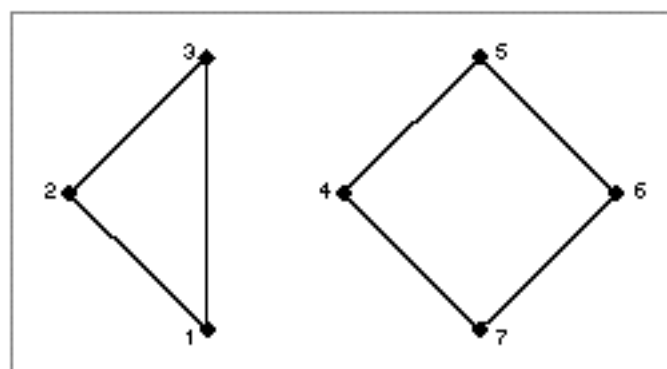


However, if the sample function had specified the `gxBreakLeftEdit` constant, as with the call

```
GXSetPolygonParts(aPolygonsShape, 4, 0,
                  (gxPolygons *) newCenterGeometry,
                  gxBreakLeftEdit);
```

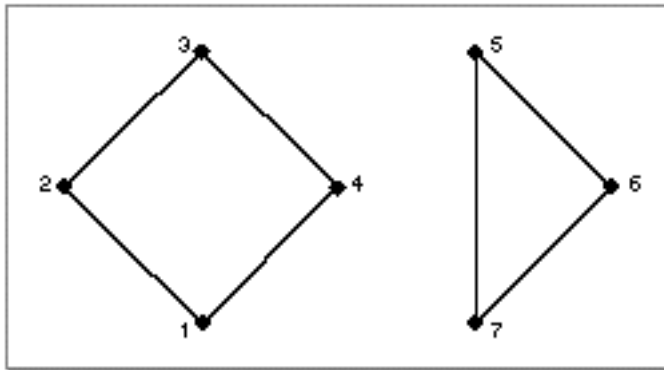
QuickDraw GX would break the resulting polygon into two contours: The `gxBreakLeftEdit` constant indicates that the polygon should be broken between the newly inserted point and the previous point, as shown in Figure 2-57.

**Figure 2-57** A polygon shape edited with the `gxBreakLeftEdit` flag set



The `gxBreakRightEdit` constant works similarly, except the break occurs between the newly inserted point and the subsequent point, as shown in Figure 2-58.

**Figure 2-58** A polygon shape edited with the `gxBreakRightEdit` flag set



You can use the `GXSetPolygonParts` function to insert a polygon geometry that contains multiple contours. In this case, the breaks that occur in the inserted geometry remain in the resulting polygon shape.

For more information about polygon geometries, see “Polygon Shapes” on page 2-22.

For more information about the `GXGetPolygonParts` and `GXSetPolygonParts` functions, see the function descriptions on page 2-144 and page 2-145.

The next two sections show more examples of editing shape geometries.

## Editing Paths Parts

---

The `GXGetPathParts` and `GXSetPathParts` functions work similarly to the `GXGetPolygonParts` and `GXSetPolygonParts` functions, which are described in the previous section.

The sample function in Listing 2-31 creates a path shape similar to the polygon shape from the previous section. Later examples in this section use this path shape to demonstrate editing path parts.

---

**Listing 2-31** Creating a path shape with two curved contours

```
void CreateTwoCurves(void)
{
    gxShape  aPathShape;

    static long twoCurveGeometry[] = {2, /* number of contours */
                                       3, /* number of points */
                                       0x40000000, /* 0100 ... */
                                       ff(100), ff(150), /* on */
                                       ff(50), ff(100), /* off */
                                       ff(100), ff(50), /* on */
                                       3, /* number of points */
                                       0x40000000, /* 0100 ... */
                                       ff(200), ff(50), /* on */
                                       ff(250), ff(100), /* off */
                                       ff(200), ff(150)}; /* on */

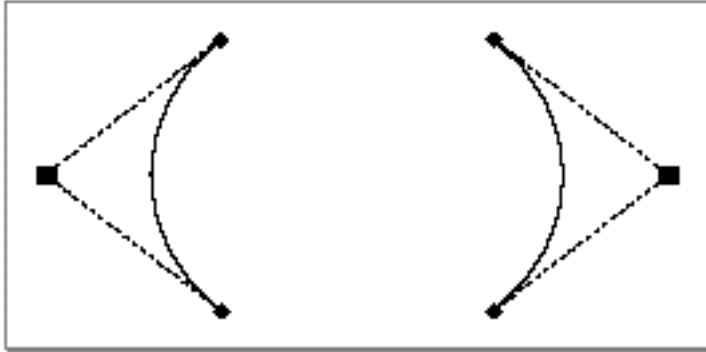
    aPathShape = GXNewPaths((gxPaths *) twoCurveGeometry);
    GXSetShapeFill(aPathShape, gxOpenFrameFill);

    GXDrawShape(aPathShape);

    GXDisposeShape(aPathShape);
}
```

The resulting path shape is shown in Figure 2-59.

**Figure 2-59** A path shape with two curved contours



You can use the `GXSetPathParts` function to replace any number of geometric points from this path shape with an arbitrary number of new geometric points. In a manner similar to the `GXSetPolygonParts` function, the `GXSetPathParts` function requires that you encapsulate the new geometric points in a path geometry. For example, to replace the top two geometric points in the path shape shown in Figure 2-59 with a single geometric point, you must first encapsulate the new geometric point in a path geometry, as with the definition

```
static long newTopGeometry[] = {1, /* number of contours */
                                1, /* number of points */
                                0x00000000, /* 0000 ... */
                                ff(150), ff(50)}; /* on curve */
```

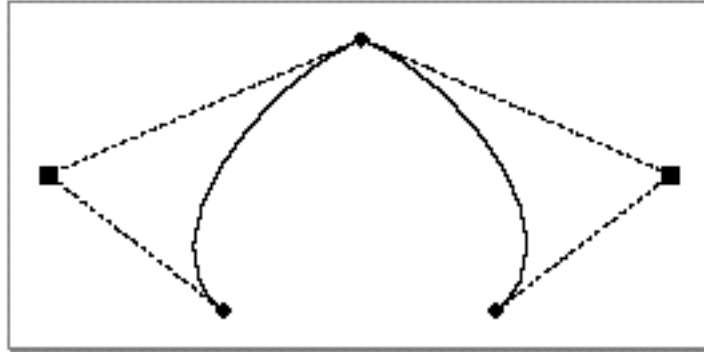
and then call the `GXSetPathParts` function:

```
GXSetPathParts(aPathsShape, 3, 2,
               (gxPaths *) newTopGeometry, gxBreakNeitherEdit);
```



The resulting path shape is shown in Figure 2-60.

**Figure 2-60** A path shape edited with `GXSetPathParts`



For more information about path geometries, see “Path Shapes” beginning on page 2-25.

For more information about the `GXGetPathParts` and `GXSetPathParts` functions, see the function descriptions on page 2-148 and page 2-149.

## Editing Shape Parts

The `GXSetShapeParts` function is more general than the `GXSetPolygonParts` and `GXSetPathParts` functions described in the previous two sections. The `GXSetShapeParts` function allows you to replace a subset of the geometric points in one shape with the geometric points in the geometry of another shape.

For example, with `GXSetShapeParts` you could replace the last three geometric points of a polygon shape with the geometry of a line shape, or you could replace the first geometric point of a path shape with the entire geometry of a polygon shape.

## Geometric Shapes

The sample function in Listing 2-32 creates a path shape with one contour. Later examples in this section use this path shape to demonstrate editing shape parts.

---

**Listing 2-32**    Creating a path shape with one contour

```
void CreatePathShape(void)
{
    gxShape  aPathShape;

    static long twoCurveGeometry[] = {1, /* number of contours */
                                       6, /* number of points */
                                       0x48000000, /* 0100 1000 */
                                       ff(100), ff(150), /* on */
                                       ff(50),  ff(100), /* off */
                                       ff(100), ff(50), /* on */
                                       ff(200), ff(50), /* on */
                                       ff(250), ff(100), /* off */
                                       ff(200), ff(150)}; /* on */

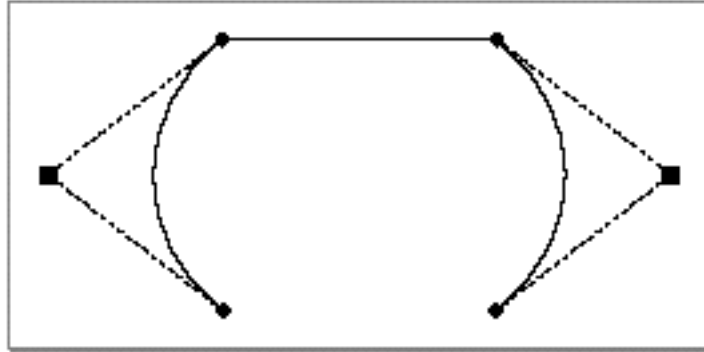
    aPathsShape = GXNewPaths((gxPaths *) twoCurveGeometry);
    GXSetShapeFill(aPathShape, gxOpenFrameFill);

    GXDrawShape(aPathShape);

    GXDisposeShape(aPathShape);
}
```

The resulting shape is shown in Figure 2-61.

**Figure 2-61** A path shape with a flat top



To insert a new geometric point in this shape using the `GXSetShapeParts` function, you must first encapsulate the new geometric point in a point shape:

```
static gxPoint newTopGeometry = {ff(150), ff(20)};
gxShape aPointShape;
```

```
aPointShape = GXNewPoint(&newTopGeometry);
```

Then you call the `GXSetShapeParts` function:

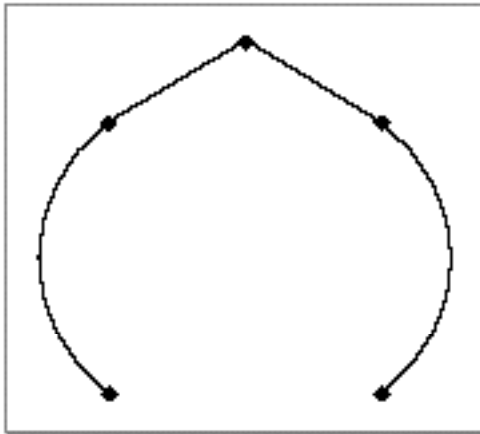
```
GXSetShapeParts(aPathsShape, 4, 0, aPointShape,
                gxBreakNeitherEdit);
```

Since you must create a shape to encapsulate the point geometry, you should dispose of this shape when you no longer need it:

```
GXDisposeShape(aPointShape);
```

The resulting path shape is shown in Figure 2-62.

**Figure 2-62** A path shape edited to have a pointy top



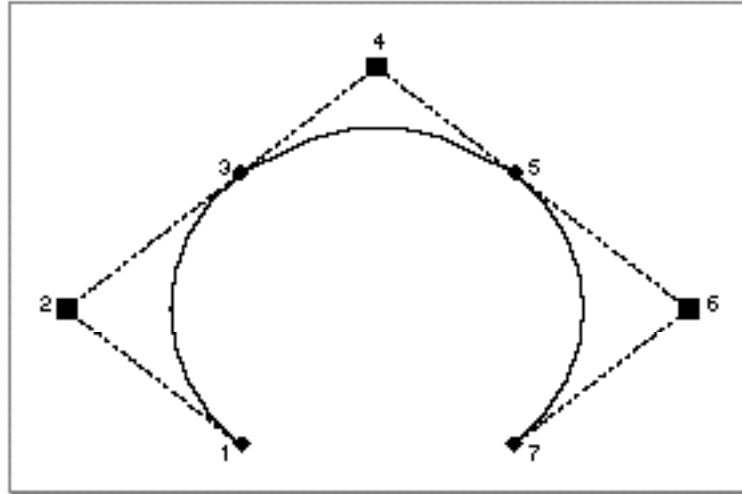
You can also use the `GXSetShapeParts` function to insert an off-curve control point in the path shape. To do this, however, you must encapsulate the new geometric point into a path shape, because only a path shape can contain a single off-curve point.

```
gxShape aSingleOffCurvePoint;
static long newTopGeometry[] = {1, /* number of contours */
                                1, /* number of points */
                                0x80000000, /* 1000 ... */
                                ff(150), ff(20)}; /* off curve */

aSingleOffCurvePoint = GXNewPaths((gxPaths *) newTopGeometry);
GXSetShapeParts(aPathsShape, 4, 0,
                aSingleOffCurvePoint, gxBreakNeitherEdit);
GXDisposeShape(aSingleOffCurvePoint);
```

The resulting path shape is shown in Figure 2-63.

**Figure 2-63** A path shape edited to have a round top



## Geometric Shapes

The `GXSetShapeParts` function allows you to edit the geometry of any shape. For example, the sample function in Listing 2-33 creates a line shape and uses `GXSetShapeParts` to change the last point.

---

**Listing 2-33** Creating a diagonal line

```
void CreateDiagonalLine(void)
{
    gxShape aLineShape;
    gxShape aPointShape;

    static gxLine lineGeometry = {ff(50), ff(50),
                                   ff(150), ff(150)};

    static gxPoint newLastPointGeometry = {ff(300), ff(150)};

    aLineShape = GXNewLine(&lineGeometry);
    GXSetShapeFill(aLineShape, gxOpenFrameFill);

    aPointShape = GXNewPoint(&newLastPointGeometry);
    GXSetShapeParts(aLineShape, 2, 1, aPointShape,
                    gxBreakNeitherEdit);
    GXDisposeShape(aPointShape);

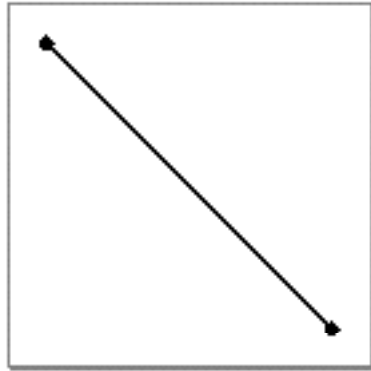
    GXDrawShape(aLineShape);

    GXDisposeShape(aLineShape);
}
```

The original line is shown in Figure 2-64.

---

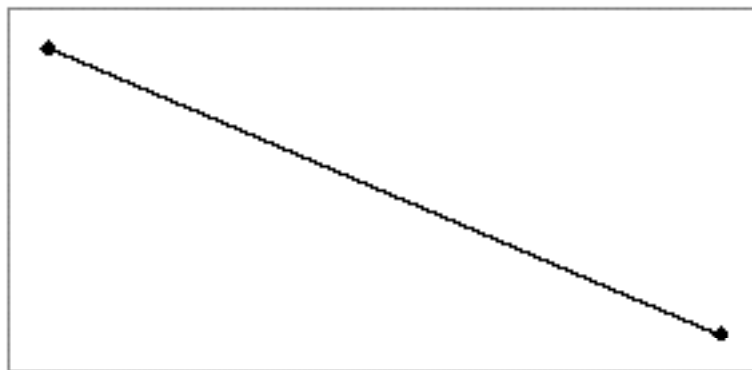
**Figure 2-64** A diagonal line



The line shape with the replaced last point is shown in Figure 2-65.

---

**Figure 2-65** An edited line



For more information about editing shape parts and the `GXSetShapeParts` function, see the function description on page 2-154.

## Applying Functions Described Elsewhere to Geometric Shapes

---

QuickDraw GX provides many functions that apply exclusively to geometric shapes. However, there are many other QuickDraw GX functions that apply to other types of shapes as well as geometric shapes.

The next two sections discuss how functions described elsewhere operate on geometric shapes. These sections are:

- n “Shape-Related Functions Applicable to Geometric Shapes,” the next section
- n “Other Functions Applicable to Geometric Shapes,” on page 2-103

### Shape-Related Functions Applicable to Geometric Shapes

---

You can apply all of the functions described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects* to geometric shapes. These functions allow you to

- n manipulate the shape object that represents geometric shapes (for example, you can copy, clone, cache, compare, and dispose of a geometric shape)
- n set the geometry, shape type, shape fill, and shape attributes of geometric shapes
- n change the style, ink, and transform objects that are associated with geometric shapes
- n manipulate the tags and owner count of the geometric shapes

Table 2-1 gives important information about geometric shapes for a subset of the functions from the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*. Functions described in that chapter that do not appear in this list exhibit the same behavior when applied to geometric shapes as they do when applied to other types of shapes.



## Geometric Shapes

**Table 2-1** Shape-related functions that exhibit special behavior with geometric shapes

Function name	Action taken
<code>GXGetDefaultShape</code>	Returns a reference to the default geometric shape of the specified type. See “The Geometric Shape Types” beginning on page 2-16 for information about the default geometric shapes.
<code>GXGetShapeFill</code>	Returns the shape fill of the shape. See “The Geometric Shape Types” beginning on page 2-16 for a discussion of which shape fills are appropriate for which geometric shapes.
<code>GXSetDefaultShape</code>	Allows you to specify the shape to copy when creating new geometric shapes. See “The Geometric Shape Types” beginning on page 2-16 for information about the default geometric shapes.
<code>GXSetShapeFill</code>	Sets the shape fill of the shape. See “The Geometric Shape Types” beginning on page 2-16 for a discussion of which shape fills are appropriate for which geometric shapes.
<code>GXSetShapeType</code>	Changes the shape type of the geometric shape and converts the shape fill and geometry as appropriate. See the rest of this section for more information about converting shape types.

When converting between geometric shape types, the behavior of the `GXSetShapeType` function depends on the new shape type. If the new shape type is the point, line or rectangle type, the new geometry is based on the bounding rectangle of the original geometry:

Old type	New type	New geometry
Any	Point	Upper-left corner of bounds
Any	Line	Line from upper-left corner to lower-left corner
Any	Rectangle	Bounding rectangle of original geometry

For examples, see “Converting Between Geometric Shape Types” beginning on page 2-65.

## Geometric Shapes

If the new shape type is the curve type, the conversion performed depends on the original shape type:

Old type	New type	New geometry
Point	Curve	New control points all set to original point
Line	Curve	First and last points remain the same; off-curve control point set equal to last point
Rectangle	Curve	First point set to original upper-left point; last point set to original lower-right point; off-curve control point set equal to last point
Polygon	Curve	New control points set to first three original control points
Path	Curve	New control points set to first three original control points

For examples, see “Converting Shapes to Curve Shapes” beginning on page 2-71.

If the new shape type is the polygon type, this function retains all of the original geometric points:

Old type	New type	New geometry
Point, line, or rectangle	Polygon	Single contour with same geometric points
Curve	Polygon	Single contour with same geometric points; the off-curve point becomes on curve
Path	Polygon	Same geometric points; all on curve (calculates approximation if curve error is not zero)

When converting a path shape to a polygon shape, this function examines the curve error of the style of the path shape. If the curve error is not zero, this function creates a polygon approximation of the original path. For more information about curve error, see the next chapter, “Geometric Styles,” in this book.

Finally, if the new shape type is the path type, the `GXSetShapeType` function retains all of the original geometry information:

Old type	New type	New geometry
Point, line, curve, or rectangle	Path	Single contour with same geometric points
Polygon	Path	Same number of contours; same geometric points; all control points remain on curve

For examples, see “Converting Shapes to Polygons and Paths” beginning on page 2-74.

## Other Functions Applicable to Geometric Shapes

---

You can apply any of the geometric operations described in Chapter 4, “Geometric Operations,” to the geometric shapes.

Geometric shapes make use of the geometric properties of their style objects. For this reason, you may apply shape-based functions (such as `GXSetShapePen`, `GXSetShapeDash`, and so on) described in Chapter 3, “Geometric Styles,” to geometric shapes.

You may also apply any of the shape-based functions in the chapter “Typographic Styles” in *Inside Macintosh: QuickDraw GX Typography* to geometric shapes. However, these functions do not affect the way geometric shapes appear when drawn.

You may apply any of the shape-based functions described in the chapter “Ink Objects” in *Inside Macintosh: QuickDraw GX Typography* to geometric shapes. These functions include `GXSetShapeColor`, `GXSetShapeTransfer`, `GXSetShapeInkAttributes`, and so on.

You may apply any of the shape-based functions described in the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Typography* to geometric shapes. These functions include `GXSetShapeClip`, `GXSetShapeMapping`, `GXSetShapeHitTest`, and so on.

## Geometric Shapes Reference

---

This section describes the data types and functions that are related to geometric shapes.

The “Data Types” section shows the structure definitions for the geometries of the geometric shapes.

The “Functions” section describes the functions that allow you to create and draw geometric shapes and functions that allow you to perform simple manipulations on shape geometries, such as replacing the entire geometry, or replacing certain points in a geometry.

Chapter 4, “Geometric Operations,” in this book describes functions that allow you to perform more advanced operations on shape geometries—operations such as inseting, intersecting, and so on.

## Data Types

---

This section describes the structures that you use when creating and manipulating geometric shapes.

You use the `gxPoint` structure when creating a point shape and when specifying geometric point positions for all of the geometric shapes.

You use the `gxLine` structure when creating a line shape.

You use the `gxCurve` structure when creating a curve shape.

You use the `gxRectangle` structure when creating a rectangle shape and when specifying the bounding rectangle of a shape.

You use the `gxPolygon` structure when specifying a single contour made up of straight lines. You use the `gxPolygons` structure when specifying multiple contours made up of straight lines.

You use the `gxPath` structure when specifying a single contour made up of straight lines and curves. You use the `gxPaths` structure when specifying multiple path contours.

## The Point Structure

---

You use the `gxPoint` structure in a number of situations; for example, to specify the geometry of a point shape, to specify the position of geometric points in the geometries of other geometric shape types, to specify a location to hit-test, to specify the position of a bitmap, and so on.

The `gxPoint` structure is defined as follows:

```
struct gxPoint {
    Fixed    x;
    Fixed    y;
};
```

### Field descriptions

<code>x</code>	A horizontal distance. Greater values of the <code>x</code> field indicate distances further to the right.
<code>y</code>	A vertical distance. Greater values of the <code>y</code> field indicate distances further down.

The location of the origin depends on the context where you use the point; for example, it might be the upper-left corner of a view port.

Notice that the `x` and `y` fields are of type `Fixed`. QuickDraw GX allows you to specify fractional coordinate positions.

For more information about coordinates and coordinate spaces, see *Inside Macintosh: QuickDraw GX Objects*.

For more information about points and point shapes, see “Point Shapes” on page 2-16.

## The Line Structure

---

You use the `gxLine` structure to specify the geometry of a line shape.

The `gxLine` structure is defined as follows:

```
struct gxLine {
    struct gxPoint first;
    struct gxPoint last;
};
```

### Field descriptions

<code>first</code>	The coordinate position where the line begins.
<code>last</code>	The coordinate position where the line ends.

Notice that the endpoints of a line are ordered—lines have an implicit direction. This direction can affect how QuickDraw GX draws a line shape, particularly when the line shape has stylistic variations.

For more information about lines and line shapes, see “Line Shapes” on page 2-17.

## The Curve Structure

---

You use the `gxCurve` structure to specify the geometry of a curve shape.

The `gxCurve` structure is defined as follows:

```
struct gxCurve {
    struct gxPoint first;
    struct gxPoint control;
    struct gxPoint last;
};
```

### Field descriptions

<code>first</code>	The coordinate position where the curve begins.
<code>control</code>	The coordinate position of the off-curve control point, which QuickDraw GX uses to determine the tangents of the curve.
<code>last</code>	The coordinate position where the curve ends.

The curve defined by these three points is a quadratic Bézier curve.

Because the geometric points that define a curve are ordered, curves have direction. The direction of a curve can affect how QuickDraw GX draws the curve shape, particularly when the curve shape has stylistic variations.

For more information about curves and curve shapes, see “Curve Shapes” on page 2-18.

## The Rectangle Structure

---

You use the `gxRectangle` structure in a variety of situations: to specify the geometry of a rectangle shape, to specify the bounding rectangle of another shape, and so on.

The `gxRectangle` structure is defined as follows:

```
struct gxRectangle {
    Fixed    left;
    Fixed    top;
    Fixed    right;
    Fixed    bottom;
};
```

### Field descriptions

<code>left</code>	Specifies the x-coordinate of the rectangle's first geometric point.
<code>top</code>	Specifies the y-coordinate of the rectangle's first geometric point.
<code>right</code>	Specifies the x-coordinate of the rectangle's last geometric point.
<code>bottom</code>	Specifies the y-coordinate of the rectangle's last geometric point.

You may specify a rectangle's geometric points in any order—the coordinates in the `left` and `top` field do not have to correspond to the rectangle's upper-left corner.

However, rectangles calculated by QuickDraw GX, such as those returned from geometric operations as described in Chapter 4, "Geometric Operations," always have their coordinates specified in the standard order.

For more information about rectangles and rectangle shapes, see "Rectangle Shapes" on page 2-20.

## Polygon Structures

---

You use the `gxPolygon` structure to specify a single polygon contour composed of straight lines.

The `gxPolygon` structure is defined as follows:

```
struct gxPolygon {
    long          vectors;
    struct gxPoint vector[gxAnyNumber];
};
```

### Field descriptions

<code>vectors</code>	The number of geometric points in the contour.
<code>vector</code>	The coordinates of the geometric points.

The array index `gxAnyNumber` indicates that the `gxPolygon` data structure is a variable-length structure—it can include any number of points.

## Geometric Shapes

The `gxPolygons` structure allows you to group multiple polygon contours together. You use this structure when specifying the geometry of a polygon shape.

The `gxPolygons` structure is defined as follows:

```
struct gxPolygons {
    long          contours;
    struct gxPolygon contour[gxAnyNumber];
};
```

**Field descriptions**

`contours`            The number of polygon contours.  
`contour`            The polygon contours.

The array index `gxAnyNumber` indicates that the `gxPolygons` data structure is also a variable-length structure—it can include any number of `gxPolygon` structures.

**Implementation Note**

In version 1.0 of QuickDraw GX, a single polygon contour can have between 1 and 32,767 geometric points. The geometry of a polygon shape can have between 0 and 32,767 polygon contours. The total size of a polygon geometry may not exceed 2,147,483,647 bytes. <sup>u</sup>

For more information about polygons and polygon shapes, see “Polygon Shapes” on page 2-22.

## Path Structures

---

You use the `gxPath` structure to specify a single contour composed of straight lines and curves.

The `gxPath` structure is defined as follows:

```
struct gxPath {
    long          vectors;
    long          controlBits[gxAnyNumber];
    struct gxPoint vector[gxAnyNumber];
};
```

**Field descriptions**

`vectors`            The number of geometric points in the contour.  
`controlBits`        Bit flags that indicate which geometric points are on curve and which are off-curve control points.  
`vector`            The coordinates of the geometric points.

The array index `gxAnyNumber` indicates that the `gxPath` data structure is a variable-length structure—it can include any number of geometric points

## Geometric Shapes

Each bit in the array specified in the `controlBits` field indicates whether a particular point in the array specified by the vector field is on curve or off curve. A value of 0 indicates that the corresponding point is on curve and a value of 1 indicates that the corresponding point is off curve.

The `gxPaths` structure allows you to group multiple path contours together. You use this data structure when specifying the geometry of a path shape.

The `gxPaths` structure is defined as follows:

```
struct gxPaths {
    long          contours;
    struct gxPath contour[gxAnyNumber];
};
```

**Field descriptions**

`contours`            The number of path contours.  
`contour`            The path contours.

The array index `gxAnyNumber` indicates that the `gxPaths` data structure is also a variable-length structure—it can include any number of path contours.

**Implementation Note**

In version 1.0 of QuickDraw GX, a single path contour can have between 0 and 32,767 geometric points. The geometry of a path shape can between 0 and 32,767 polygon contours. The total size of a path geometry may not exceed 2,147,483,647 bytes. <sup>u</sup>

For more information about paths and path shapes, see “Path Shapes” on page 2-25.

## Functions

---

This section describes the functions available for

- n creating new geometric shapes
- n manipulating the geometries of geometric shapes
- n editing parts of shape geometries
- n drawing geometric shapes

Chapter 4, “Geometric Operations,” contains information about more sophisticated functions for manipulating shape geometries.



For information about creating, drawing, and manipulating bitmap shapes, see Chapter 5, “Bitmap Shapes.”

For information about creating, drawing, and manipulating picture shapes, see Chapter 6, “Picture Shapes.”

For information about getting and setting the default geometric shapes and information about manipulating shape type and shape fill, see the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*. For information about hit-testing geometric shapes, see the chapter “Transform Objects” also in that book.

For information about creating, drawing, and manipulating typographic shapes, see *Inside Macintosh: QuickDraw GX Typography*.

## Creating Geometric Shapes

---

QuickDraw GX provides a number of ways for you to create a new shape.

The functions described in this section allow you to specify a shape’s initial geometry when creating the shape. For example, the `GXNewShapeVector` function allows you to specify a shape type and an array of values. The function creates a new shape of the specified type and uses the array of values to initialize the new shape’s geometry.

The `GXNewPoint`, `GXNewLine`, `GXNewCurve`, `GXNewRectangle`, `GXNewPolygons`, and `GXNewPaths` functions all create a new shape of a specific type. These functions allow you to specify the shape’s initial geometry.

You can also use the `GXNewShape` function to create shapes. This function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*, allows you to create a shape by specifying only the shape type; the geometry of the new shape is set to its initial state—all geometric points are (0.0, 0.0) and polygons and paths have 0 contours. You can customize the shape’s geometry using the functions described in “Getting and Setting Shape Geometries” beginning on page 2-119.

## GXNewShapeVector

---

You can use the `GXNewShapeVector` function to create a new shape of any type.

```
void GXNewShapeVector(gxShapeType aType, const Fixed vector[]);
```

`aType`            A reference to the shape whose geometry you want to change.

`vector`           An array of fixed-point values to use as the new geometry.

*function result* A reference to the new shape.

## Geometric Shapes

## DESCRIPTION

The `GXNewShapeVector` function copies the default shape of the shape type specified by the `aType` parameter, sets the owner count of the new shape to 1, initializes its geometry with the values in the `vector` parameter, and returns a reference to it as the function result.

Although this function creates a copy of the default shape, it does not create a copy of the default shape's style, ink, or transform. The new shape returned by this function contains references to same style, ink, and transform as the default shape. You can change the style using functions from Chapter 3, "Geometric Styles," and you can change the style, ink, and transform using functions from *Inside Macintosh: QuickDraw GX Objects*.

You may pass any number of values in the `vector` array; the `GXNewShapeVector` function traverses this array as necessary to initialize the new shape's geometry. If you pass too few values in this parameter, the function posts the warning `extra_data_passed_was_ignored`.

If you specify a shape type that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Creates a bitmap shape; expects the vector array to contain values corresponding to the fields of a bitmap structure
picture	Creates a picture shape with no overriding styles, inks, or transforms; expects the vector array to contain an array of shape references
text	Posts the error <code>graphic_type_does_not_contain_points</code>
glyph	Posts the error <code>graphic_type_does_not_contain_points</code>
layout	Posts the error <code>graphic_type_does_not_contain_points</code>

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_is_nil</code>	
<code>number_of_points_exceeds_implementation_limit</code>	
<code>number_of_contours_exceeds_implementation_limit</code>	
<code>size_of_polygon_exceeds_implementation_limit</code>	
<code>size_of_path_exceeds_implementation_limit</code>	
<code>illegal_type_for_shape</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)
<code>graphic_type_does_not_contain_points</code>	(debugging version)

**Warnings**

`extra_data_passed_was_ignored`

**SEE ALSO**

For general information about each type of geometry, see “About Geometric Shapes” on page 2-5. For specific definitions of each type of geometry, see the section “Data Types” beginning on page 2-104.

For information about related functions, see the descriptions of the `GXNewPoint`, `GXNewLine`, `GXNewCurve`, `GXNewRectangle`, `GXNewPolygons`, and `GXNewPaths` functions on page 2-111 through page 2-119.

**GXNewPoint**

---

You can use the `GXNewPoint` function to create a new point shape and initialize its geometry.

```
gxShape GXNewPoint(const gxPoint *data);
```

`data`                      A pointer to the initial point geometry.

*function result*    A reference to the new point shape.

**DESCRIPTION**

The `GXNewPoint` function creates a copy of the default point shape, sets the owner count of the copy to 1, initializes its geometry with the values in the `data` parameter, and returns a reference to it as the function result.

Although this function creates a copy of the default point shape, it does not create a copy of the default point’s style, ink, or transform objects. The new point shape returned by this function contains references to the same style, ink, and transform as the default point shape.

**SPECIAL CONSIDERATIONS**

If no error occurs, the `GXNewPoint` function creates a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about creating and disposing of shapes.

If an error occurs, this function returns `nil` as the function result.

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`  
`parameter_is_nil`            (debugging version)

## SEE ALSO

For an example that uses this function, see “Creating and Drawing Points” beginning on page 2-29.

For a discussion of points and the default point shape, see “Point Shapes” on page 2-16.

For a description of the `gxPoint` structure, see page 2-104.

To create a new point shape without specifying an initial geometry, see the description of the `GXNewShape` function in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To set the geometry of an existing point shape, see the description of the `GXSetPoint` function on page 2-122.

To draw a point geometry, see the description of `GXDrawPoint` on page 2-158. To draw a point shape, see the description of `GXDrawShape` in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXNewLine

---

You can use the `GXNewLine` function to create a new line shape and initialize its geometry.

```
gxShape GXNewLine(const gxLine *data);
```

`data`            A pointer to the initial line geometry.

*function result* A reference to the new line shape.

## DESCRIPTION

The `GXNewLine` function creates a copy of the default line shape, sets the owner count of the copy to 1, initializes its geometry with the values in the `data` parameter, and returns a reference to it as the function result.

Although this function creates a copy of the default line shape, it does not create a copy of the default line’s style, ink, or transform objects. The new line shape returned by this function contains references to same style, ink, and transform as the default line shape.

## SPECIAL CONSIDERATIONS

If no error occurs, the `GXNewLine` function creates a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about creating and disposing of shapes.

If an error occurs, this function returns `nil` as the function result.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

out\_of\_memory  
parameter\_is\_nil (debugging version)

## SEE ALSO

For an example that uses this function, see “Creating and Drawing Lines” beginning on page 2-36.

For a discussion of lines and the default line shape, see “Line Shapes” on page 2-17.

For a description of the `gxLine` structure, see page 2-105.

To create a new line shape without specifying an initial geometry, see the description of the `GXNewShape` function in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To set the geometry of an existing line shape, see the description of the `GXSetLine` function on page 2-124.

To draw a line geometry without creating a line shape, see the description of `GXDrawLine` on page 2-158. To draw a line shape, see the description of `GXDrawShape` in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

**GXNewCurve**

---

You can use the `GXNewCurve` function to create a new curve shape and initialize its geometry.

```
gxShape GXNewCurve(const gxCurve *data);
```

`data`            A pointer to the initial curve geometry.

*function result*   A reference to the new curve shape.

**DESCRIPTION**

The `GXNewCurve` function creates a copy of the default curve shape, sets the owner count of the copy to 1, initializes its geometry with the values in the `data` parameter, and returns a reference to it as the function result.

Although this function creates a copy of the default curve shape, it does not create a copy of the default curve’s style, ink, or transform objects. The new curve shape returned by this function contains references to same style, ink, and transform as the default curve shape.

**SPECIAL CONSIDERATIONS**

If no error occurs, the `GXNewCurve` function creates a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about creating and disposing of shapes.

If an error occurs, this function returns `nil` as the function result.

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`  
`parameter_is_nil` (debugging version)

**SEE ALSO**

For an example that uses this function, see “Creating and Drawing Curves” beginning on page 2-41.

For a discussion of curves and the default curve shape, see “Curve Shapes” beginning on page 2-18.

For a description of the `gxCurve` structure, see page 2-105.

To create a new curve shape without specifying an initial geometry, see the description of the `GXNewShape` function in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To set the geometry of an existing curve shape, see the description of the `GXSetCurve` function on page 2-126.

To draw a curve geometry without creating a curve shape, see the description of `GXDrawCurve` on page 2-159. To draw a curve shape, see the description of `GXDrawShape` in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

**GXNewRectangle**

---

You can use the `GXNewRectangle` function to create a new rectangle shape and initialize its geometry.

```
gxShape GXNewRectangle(const gxRectangle *data);
```

`data`            A pointer to the initial rectangle geometry.

*function result* A reference to the new rectangle shape.

**DESCRIPTION**

The `GXNewRectangle` function creates a copy of the default rectangle shape, sets the owner count of the copy to 1, initializes its geometry with the values in the `data` parameter, and returns a reference to it as the function result.

Although this function creates a copy of the default rectangle shape, it does not create a copy of the default rectangle's style, ink, or transform objects. The new rectangle shape returned by this function contains references to same style, ink, and transform as the default rectangle shape.

**SPECIAL CONSIDERATIONS**

If no error occurs, the `GXNewRectangle` function creates a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about creating and disposing of shapes.

If an error occurs, this function returns `nil` as the function result.

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`  
`parameter_is_nil` (debugging version)

**SEE ALSO**

For an example that uses this function, see “Creating and Drawing Rectangles” beginning on page 2-43.

For a discussion of rectangles and the default rectangle shape, see “Rectangle Shapes” beginning on page 2-20.

For a description of the `gxRectangle` structure, see page 2-106.

To create a new rectangle shape without specifying an initial geometry, see the description of the `GXNewShape` function in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To set the geometry of an existing rectangle shape, see the description of the `GXSetRectangle` function on page 2-129.

To draw a rectangle geometry without creating a rectangle shape, see the description of `GXDrawRectangle` on page 2-160. To draw a rectangle shape, see the description of `GXDrawShape` in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXNewPolygons

---

You can use the `GXNewPolygons` function to create a new polygon shape and initialize its geometry.

```
gxShape GXNewPolygons(const gxPolygons *data);
```

`data`                    A pointer to the initial polygon geometry.

*function result*    A reference to the new polygon shape.

### DESCRIPTION

The `GXNewPolygons` function creates a copy of the default polygon shape, sets the owner count of the copy to 1, initializes its geometry with the values in the `data` parameter, and returns a reference to it as the function result. If you specify `nil` for the `data` parameter, this function returns a polygon shape with no polygon contours.

Although this function creates a copy of the default polygon shape, it does not create a copy of the default polygon's style, ink, or transform objects. The new polygon shape returned by this function contains references to same style, ink, and transform as the default polygon shape.

#### Implementation Note

In version 1.0 of QuickDraw GX, the total size of a polygon geometry may not exceed 2,147,483,647 bytes. If the size of the data you provide in the `data` parameter exceeds this limit, the `GXNewPolygons` function posts a `size_of_polygon_exceeds_implementation_limit` error. <sup>u</sup>

### SPECIAL CONSIDERATIONS

If no error occurs, the `GXNewPolygons` function creates a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about creating and disposing of shapes.

If an error occurs, this function returns `nil` as the function result.



## ERRORS, WARNINGS, AND NOTICES

**Errors**

```

out_of_memory
number_of_points_exceeds_implementation_limit
number_of_contours_exceeds_implementation_limit
size_of_polygon_exceeds_implementation_limit
count_is_less_than_one

```

(debugging version)

## SEE ALSO

For an example that uses this function, see “Creating and Drawing Polygons” beginning on page 2-45.

For a discussion of polygons and the default polygon shape, see “Polygon Shapes” beginning on page 2-22.

For a description of the `gxPolygons` structure, see page 2-106.

To create a new polygon shape without specifying an initial geometry, see the description of the `GXNewShape` function in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To set the geometry of an existing polygon shape, see the description of the `GXSetPolygons` function on page 2-131.

To draw a polygon geometry without creating a polygon shape, see the description of `GXDrawPolygons` on page 2-161. To draw a polygon shape, see the description of `GXDrawShape` in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXNewPaths

---

You can use the `GXNewPaths` function to create a new path shape and initialize its geometry.

```
gxShape GXNewPaths(const gxPaths *data);
```

`data`            A pointer to the initial path geometry.

*function result*   A reference to the new path shape.

**DESCRIPTION**

The `GXNewPaths` function creates a copy of the default path shape, sets the owner count of the copy to 1, initializes its geometry with the values in the `data` parameter, and returns a reference to it as the function result. If you specify `nil` for the `data` parameter, this function returns a path shape with no path contours.

Although this function creates a copy of the default path shape, it does not create a copy of the default path shape's style, ink, or transform objects. The new path shape returned by this function contains references to same style, ink, and transform as the default path shape.

**Implementation Limit**

In version 1.0 of QuickDraw GX, the total size of a path geometry may not exceed 2,147,483,647 bytes. If the size of the data you provide in the `data` parameter exceeds this limit, the `GXNewPaths` function posts a `size_of_path_exceeds_implementation_limit` error. u

**SPECIAL CONSIDERATIONS**

If no error occurs, the `GXNewPaths` function creates a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about creating and disposing of shapes.

If an error occurs, this function returns `nil` as the function result.

**ERRORS, WARNINGS, AND NOTICES****Errors**

<code>out_of_memory</code>	
<code>number_of_points_exceeds_implementation_limit</code>	
<code>number_of_contours_exceeds_implementation_limit</code>	
<code>size_of_path_exceeds_implementation_limit</code>	
<code>count_is_less_than_one</code>	(debugging version)

**SEE ALSO**

For an example that uses this function, see “Creating and Drawing Paths” beginning on page 2-55.

For a discussion of paths and the default path shape, see “Path Shapes” beginning on page 2-25.

For a description of the `gxPaths` structure, see page 2-107.

To create a new path shape without specifying an initial geometry, see the description of the `GXNewShape` function in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To set the geometry of an existing path shape, see the description of the `GXSetPaths` function on page 2-133.

To draw a path geometry without creating a path shape, see the description of `GXDrawPaths` on page 2-162. To draw a path shape, see the description of `GXDrawShape` in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## Getting and Setting Shape Geometries

---

The geometry property of geometric shapes contains the geometric points that define the shape. The geometries of polygon shapes and path shapes also contain some additional information, such as the number of separate contours, how many geometric points in each contour, and (for paths) which geometric points are on curve and which are off-curve control points.

For general information about each type of geometry, see “About Geometric Shapes” beginning on page 2-5. For specific definitions of each type of geometric structure, see the section “Data Types” beginning on page 2-104.

The `GXSetShapeVector` function allows you to change the geometry of any shape. With this function, you specify a shape and an array of values. The function replaces the geometry of the specified shape with the values in the array. This function works for other shape types as well as geometric shapes.

The `GXGetPoint`, `GXGetLine`, `GXGetCurve`, `GXGetRectangle`, `GXGetPolygons`, and `GXGetPaths` functions each return the geometry of a specific type of shape.

The `GXSetPoint`, `GXSetLine`, `GXSetCurve`, `GXSetRectangle`, `GXSetPolygons`, and `GXSetPaths` functions each replace the geometry of a specific type of shape.

## GXSetShapeVector

---

You can use the `GXSetShapeVector` function to change the geometry of an existing shape.

```
void GXSetShapeVector(gxShape target, const Fixed vector[]);
```

**target**            A reference to the shape whose geometry you want to change.

**data**             An array of fixed-point values to use as the new geometry.

### DESCRIPTION

The `GXSetShapeVector` function replaces the geometry of the `target` shape with a new geometry, which it creates by traversing the `vector` array. The length of the vector array that you supply depends on shape type of the `target` shape; for example, if the `target` shape is a point, you should provide a vector array with two `Fixed` values; if the `target` shape is a line, you should provide four `Fixed` values, and so on.

## Geometric Shapes

Although this function creates a copy of the default shape, it does not create a copy of the default shape's style, ink, or transform. The new shape returned by this function contains references to same style, ink, and transform as the default shape. You can change the style using functions from Chapter 3, "Geometric Styles," and you can change the style, ink, and transform using functions from *Inside Macintosh: QuickDraw GX Objects*.

You may pass any number of values in the `vector` array; the `GXNewShapeVector` function traverses this array as necessary to initialize the new shape's geometry. If you pass too few values in this parameter, the function posts the warning `extra_data_passed_was_ignored`.

If you specify a shape type that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
<code>bitmap</code>	Sets the target shape to be a bitmap shape; expects the vector array to contain values corresponding to the fields of a bitmap structure
<code>picture</code>	Sets the target shape to be a picture shape with no overriding styles, inks, or transforms; expects the vector array to contain an array of shape references
<code>text</code>	Posts the error <code>graphic_type_does_not_contain_points</code>
<code>glyph</code>	Posts the error <code>graphic_type_does_not_contain_points</code>
<code>layout</code>	Posts the error <code>graphic_type_does_not_contain_points</code>

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_is_nil</code>	
<code>number_of_points_exceeds_implementation_limit</code>	
<code>number_of_contours_exceeds_implementation_limit</code>	
<code>size_of_polygon_exceeds_implementation_limit</code>	
<code>size_of_path_exceeds_implementation_limit</code>	
<code>illegal_type_for_shape</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)
<code>graphic_type_does_not_contain_points</code>	(debugging version)

**Warnings**

`extra_data_passed_was_ignored`

**SEE ALSO**

For general information about each type of geometry, see “About Geometric Shapes” on page 2-5. For specific definitions of each type of geometry, see the section “Data Types” beginning on page 2-104.

For information about related functions, see the descriptions of the `GXSetPoint`, `GXSetLine`, `GXSetCurve`, `GXSetRectangle`, `GXSetPolygons`, and `GXSetPaths` functions on page 2-122 through page 2-135.

**GXGetPoint**

---

You can use the `GXGetPoint` function to determine the geometry of an existing point shape.

```
gxPoint *GXGetPoint(gxShape source, gxPoint *data);
```

**source**            A reference to the point shape whose geometry you want to determine.

**data**             A pointer to a `gxPoint` structure. The function copies the source shape’s geometry into this structure.

*function result*   A pointer to a copy of the source shape’s geometry.

**DESCRIPTION**

The `GXGetPoint` function copies the geometry information from the source point shape into the `gxPoint` structure pointed to by the `data` parameter. As a convenience, this function also returns a pointer to the point geometry as the function result.

If the source shape is not a point shape, this function posts the error code `illegal_type_for_shape`.

You must pass a pointer to a `gxPoint` structure in the `data` parameter—if you pass `nil` for this parameter, the function posts the error code `parameter_is_nil`.

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`

`shape_is_nil`

`illegal_type_for_shape`            (debugging version)

`parameter_is_nil`                 (debugging version)

**SEE ALSO**

For general information about point geometries, see “Point Shapes” on page 2-16.

For the definition of the `gxPoint` structure, see page 2-104.

To create a new point shape, use the `GXNewPoint` function, which is described on page 2-111.

To change the geometry of an existing point shape, use the `GXSetPoint` function, which is described in the next section.

To draw a point geometry without creating a point shape, use the `GXDrawPoint` function, which is described on page 2-158. To draw a point shape, use the `GXDrawShape` function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

**GXSetPoint**

---

You can use the `GXSetPoint` function to change the geometry of an existing point shape.

```
void GXSetPoint(gxShape target, const gxPoint *data);
```

`target`            A reference to the point shape whose geometry you want to change.

`data`             A pointer to the new point geometry.

**DESCRIPTION**

The `GXSetPoint` function copies the geometry information from the `data` parameter into the geometry property of the target point shape. If the target shape is not a point shape, this function replaces the target shape with a point shape and sets the shape fill to open-frame fill.

You must provide a pointer to a `gxPoint` structure in the `data` parameter—if you pass `nil` for the `data` parameter, the function posts the error `parameter_is_nil`.

If the target shape is locked (that is, its `gxLockedShape` shape attribute is set), this function posts the error `shape_access_not_allowed`.

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`

`shape_is_nil`

`parameter_is_nil`

`shape_access_not_allowed`

(debugging version)

(debugging version)

**SEE ALSO**

For general information about point geometries, see “Point Shapes” on page 2-16.

For the definition of the `gxPoint` structure, see page 2-104.

To create a new point shape, use the `GXNewPoint` function, which is described on page 2-111.

To examine the geometry of an existing point shape, use the `GXGetPoint` function, which is described on page 2-121.

To draw a point geometry without creating a point shape, use the `GXDrawPoint` function, which is described on page 2-158. To draw a point shape, use the `GXDrawShape` function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

**GXGetLine**

---

You can use the `GXGetLine` function to determine the geometry of an existing line shape.

```
gxLine *GXGetLine(gxShape source, gxLine *data);
```

**source**        A reference to the line shape whose geometry you want to determine.

**data**            A pointer to a `gxLine` structure. The function copies the source shape's geometry into this structure.

*function result*   A pointer to a copy of the source shape's geometry.

**DESCRIPTION**

The `GXGetLine` function copies the geometry information from the source line shape into the `gxLine` structure pointed to by the `data` parameter. As a convenience, this function also returns a pointer to the line geometry as the function result.

If the source shape is not a line shape, this function posts the error code `illegal_type_for_shape`.

You must pass a pointer to a `gxLine` structure in the `data` parameter—if you pass `nil` for this parameter, the function posts the error code `parameter_is_nil`.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>illegal_type_for_shape</code>	(debugging version)
<code>parameter_is_nil</code>	(debugging version)

## SEE ALSO

For general information about line geometries, see “Line Shapes” on page 2-17.

For the definition of the `gxLine` structure, see page 2-105.

To create a new line shape, use the `GXNewLine` function, which is described on page 2-112.

To change the geometry of an existing line shape, use the `GXSetLine` function, which is described in the next section.

To draw a line geometry without creating a line shape, use the `GXDrawLine` function, which is described on page 2-158. To draw a line shape, use the `GXDrawShape` function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

**GXSetLine**

---

You can use the `GXSetLine` function to change the geometry of a line shape.

```
void GXSetLine(gxShape target, const gxLine *data);
```

`target`      A reference to the line shape whose geometry you want to change.

`data`        A pointer to the new line geometry.

## DESCRIPTION

The `GXSetLine` function copies the geometry information from the `data` parameter into the geometry property of the target line shape. If the target shape is not a line shape, this function replaces the target shape with a line shape and sets the shape fill to open-frame fill.

You must provide a pointer to a `gxLine` structure in the `data` parameter—if you pass `nil` for this parameter, the function posts the error code `parameter_is_nil`.

If the target shape is locked (that is, its `gxLockedShape` shape attribute is set), this function posts the error `shape_access_not_allowed`.



## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)

## SEE ALSO

For general information about line geometries, see “Line Shapes” on page 2-17.

For the definition of the `gxLine` structure, see page 2-105.

To create a new line shape, use the `GXNewLine` function, which is described on page 2-112.

To examine the geometry of an existing line shape, use the `GXGetLine` function, which is described on page 2-123.

To draw a line geometry without creating a line shape, use the `GXDrawLine` function, which is described on page 2-158. To draw a line shape, use the `GXDrawShape` function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

**GXGetCurve**

---

You can use the `GXGetCurve` function to determine the geometry of an existing curve shape.

```
gxCurve *GXGetCurve(gxShape source, gxCurve *data);
```

**source**      A reference to the curve shape whose geometry you want to determine.

**data**        A pointer to a `gxCurve` structure. The function copies the source shape's geometry into this structure.

**function result** A pointer to a copy of the source shape's geometry.

## DESCRIPTION

The `GXGetCurve` function copies the geometry information from the source curve shape into the `gxCurve` structure pointed to by the `data` parameter. As a convenience, this function also returns a pointer to the curve geometry as the function result.

## Geometric Shapes

If the source shape is not a curve shape, this function posts the error code `illegal_type_for_shape`.

You must pass a pointer to a `gxCurve` structure in the `data` parameter—if you pass `nil` for this parameter, the function posts the error code `parameter_is_nil`.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>illegal_type_for_shape</code>	(debugging version)
<code>parameter_is_nil</code>	(debugging version)

## SEE ALSO

For general information about curve geometries, see “Curve Shapes” on page 2-18.

For the definition of the `gxCurve` structure, see page 2-105.

To create a new curve shape, use the `GXNewCurve` function, which is described on page 2-113.

To change the geometry of an existing curve shape, use the `GXSetCurve` function, which is described in the next section.

To draw a curve geometry without creating a curve shape object, use the `GXDrawCurve` function, which is described on page 2-159. To draw a curve shape, use the `GXDrawShape` function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

**GXSetCurve**

---

You can use the `GXSetCurve` function to change the geometry of a curve shape.

```
void GXSetCurve(gxShape target, const gxCurve *data);
```

<code>target</code>	A reference to the curve shape whose geometry you want to change.
<code>data</code>	A pointer to the new curve geometry.

**DESCRIPTION**

The `GXSetCurve` function copies the geometry information from the `data` parameter into the geometry property of the target shape. If the target shape is not a curve shape, this function replaces the target shape with a curve shape and sets the shape fill to open-frame fill.

You must provide a pointer to a `gxCurve` structure in the `data` parameter—if you pass `nil` for this parameter, the function posts the error code `parameter_is_nil`.

If the target shape is locked (that is, its `gxLockedShape` shape attribute is set), this function posts the error `shape_access_not_allowed`.

#### ERRORS, WARNINGS, AND NOTICES

##### Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)

#### SEE ALSO

For general information about curve geometries, see “Curve Shapes” on page 2-18.

For the definition of the `gxCurve` structure, see page 2-105.

To create a new curve shape, use the `GXNewCurve` function, which is described on page 2-113.

To examine the geometry of an existing curve shape, use the `GXGetCurve` function, which is described on page 2-125.

To draw a curve geometry without creating a curve shape, use the `GXDrawCurve` function, which is described on page 2-159. To draw a curve shape, use the `GXDrawShape` function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXGetRectangle

---

You can use the `GXGetRectangle` function to determine the geometry of an existing rectangle shape.

```
gxRectangle *GXGetRectangle(gxShape source, gxRectangle *data);
```

**source**      A reference to the rectangle shape whose geometry you want to determine.

**data**        A pointer to a `gxRectangle` structure. The function copies the source shape's geometry into this structure.

*function result*   A pointer to a copy of the source shape's geometry.

## Geometric Shapes

## DESCRIPTION

The `GXGetRectangle` function copies the geometry information from the source rectangle shape into the `gxRectangle` data structure pointed to by the `data` parameter. As a convenience, this function also returns a pointer to the rectangle geometry as the function result.

If the source shape is not a rectangle shape, this function posts the error code `illegal_type_for_shape`.

You must pass a pointer to a `gxRectangle` structure in the `data` parameter—if you pass `nil` for this parameter, the function posts the error code `parameter_is_nil`.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

`out_of_memory`

`shape_is_nil`

`illegal_type_for_shape` (debugging version)

`parameter_is_nil` (debugging version)

## SEE ALSO

For general information about rectangle geometries, see “Rectangle Shapes” on page 2-20.

For the definition of the `gxRectangle` structure, see page 2-106.

To create a new rectangle shape, use the `GXNewRectangle` function, which is described on page 2-114.

To determine the bounding rectangle of a rectangle shape, use the `GXGetShapeBounds` function, which is described in the chapter, “Geometric Operations,” in this book. (The result of the `GXGetShapeBounds` function is an ordered rectangle. Therefore, the result of this function may differ from the geometry of the shape you pass in, even if that shape is a rectangle.)

To change the geometry of an existing rectangle shape, use the `GXSetRectangle` function, which is described in the next section.

To draw a rectangle geometry without creating a rectangle shape, use the `GXDrawRectangle` function, which is described on page 2-160. To draw a rectangle shape, use the `GXDrawShape` function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXSetRectangle

---

You can use the `GXSetRectangle` function to change the geometry of a rectangle shape.

```
void GXSetRectangle(gxShape target, const gxRectangle *data);
```

`target`      A reference to the rectangle shape whose geometry you want to change.

`data`        A pointer to the new rectangle geometry.

### DESCRIPTION

The `GXSetRectangle` function copies the geometry information from the `data` parameter into the geometry property of the target shape. If the target shape is not a rectangle shape, this function replaces the target shape with a rectangle shape and sets the shape fill to closed-frame fill if it was originally open-frame fill.

If the target shape is not a rectangle shape, this function posts the error code `illegal_type_for_shape`.

You must provide a pointer to a `gxRectangle` structure in the `data` parameter—if you pass `nil` for this parameter, the function posts the error code `parameter_is_nil`.

If the target shape is locked (that is, its `gxLockedShape` shape attribute is set), this function posts the error `shape_access_not_allowed`.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)

### SEE ALSO

For general information about rectangle geometries, see “Rectangle Shapes” on page 2-20.

For the definition of the `gxRectangle` structure, see page 2-106.

To create a new rectangle shape, use the `GXNewRectangle` function, which is described on page 2-114.

To examine the geometry of an existing rectangle shape, use the `GXGetRectangle` function, which is described on page 2-127.

To draw a rectangle geometry without creating a rectangle shape, use the `GXDrawRectangle` function, which is described on page 2-160. To draw a rectangle shape, use the `GXDrawShape` function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXGetPolygons

---

You can use the `GXGetPolygons` function to determine the geometry of a polygon shape.

```
long GXGetPolygons(gxShape source, gxPolygons *data);
```

`source`      A reference to the polygon shape whose geometry you want to determine.

`data`          A pointer to a `gxPolygons` data structure. The function copies the source shape's geometry into this structure.

*function result*   The length in bytes of the source shape's geometry.

### DESCRIPTION

The `GXGetPolygons` function copies the geometry information from the source polygon shape into the `gxPolygons` structure pointed to by the `data` parameter. As the function result, this function returns the length in bytes of the polygon geometry.

If the source shape is not a polygon shape, this function posts the error code `illegal_type_for_shape`.

You may pass `nil` for the `data` parameter. In this case, the `GXGetPolygons` function still returns the length of the data as the function result, but it does not return the actual data in the `data` parameter.

Typically, to use this function, you go through the following steps:

1. Determine the length of the polygon data by calling this function, passing `nil` for the `data` parameter.
2. Allocate enough memory to hold the polygon data.
3. Call this function again, passing a pointer to the allocated memory in the `data` parameter.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`

`shape_is_nil`

`illegal_type_for_shape`      (debugging version)

### SEE ALSO

For general information about polygon geometries, see “Polygon Shapes” on page 2-22.

For the definition of the `gxPolygons` structure, see page 2-106.

To create a new polygons shape, use the `GXNewPolygons` function, which is described on page 2-116.

To change the geometry of an existing polygon shape, use the `GXSetPolygons` function, which is described in the next section.

To draw a polygon geometry without creating a polygon shape, use the `GXDrawPolygons` function, which is described on page 2-161. To draw a polygons shape, use the `GXDrawShape` function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXSetPolygons

---

You can use the `GXSetPolygons` function to change the geometry of a polygon shape.

```
void GXSetPolygons(gxShape target, const gxPolygons *data);
```

`target`        A reference to the polygon shape whose geometry you want to change.

`data`         A pointer to the new polygon geometry.

### DESCRIPTION

The `GXSetPolygons` function copies the geometry information from the `data` parameter into the geometry property of the target polygon shape. If the target shape is not a polygon shape, this function replaces the target shape with a polygon shape.

If you pass `nil` for the `data` parameter, the function sets the polygon shape to have zero contours.

If the target shape is locked (that is, its `gxLockedShape` shape attribute is set), this function posts the error `shape_access_not_allowed`.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`

`shape_is_nil`

`number_of_points_exceeds_implementation_limit`

`number_of_contours_exceeds_implementation_limit`

`size_of_polygon_exceeds_implementation_limit`

`count_is_less_than_one`

`shape_access_not_allowed`

(debugging version)

(debugging version)

## SEE ALSO

For general information about polygon geometries, see “Polygon Shapes” on page 2-22.

For the definition of the `gxPolygons` structure, see page 2-106.

To create a new polygon shape, use the `GXNewPolygons` function, which is described on page 2-116.

To examine the geometry of an existing polygon shape, use the `GXGetPolygons` function, which is described on page 2-130.

To draw a polygon geometry without creating a polygon shape, use the `GXDrawPolygons` function, which is described on page 2-161. To draw a polygon shape, use the `GXDrawShape` function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXGetPaths

---

You can use the `GXGetPaths` function to determine the geometry of a path shape.

```
long GXGetPaths(gxShape source, gxPaths *data);
```

**source**        A reference to the path shape whose geometry you want to determine.

**data**         A pointer to a `gxPaths` structure. The function copies the source shape’s geometry into this structure.

*function result*   The length in bytes of the source shape’s geometry.

## DESCRIPTION

The `GXGetPaths` function copies the geometry information from the source path shape into the `gxPaths` structure pointed to by the `data` parameter. As the function result, this function returns the length in bytes of the path geometry.

If the source shape is not a path shape, this function posts the error code `illegal_type_for_shape`.

You may pass `nil` for the `data` parameter. In this case, the `GXGetPaths` function still returns the length of the data, but it does not return the actual data in the `data` parameter.



## Geometric Shapes

Typically, to use this function, you go through the following steps:

1. Determine the length of the path data by calling this function, passing `nil` for the `data` parameter.
2. Allocate enough memory to hold the path data.
3. Call this function again, passing a pointer to the allocated memory in the `data` parameter.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

`out_of_memory`  
`shape_is_nil`  
`illegal_type_for_shape` (debugging version)

## SEE ALSO

For general information about path geometries, see “Path Shapes” on page 2-25.

For the definition of the `gxPaths` structure, see page 2-107.

To create a new path shape, use the `GXNewPaths` function, which is described on page 2-117.

To change the geometry of an existing path shape, use the `GXSetPaths` function, which is described in the next section.

To draw a path geometry without creating a path shape, use the `GXDrawPaths` function, which is described on page 2-162. To draw a path shape, use the `GXDrawShape` function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXSetPaths

---

You can use the `GXSetPaths` function to change the geometry of a path shape.

```
void GXSetPaths(gxShape target, const gxPaths *data);
```

<code>target</code>	A reference to the path shape whose geometry you want to change.
<code>data</code>	A pointer to new path geometry.

## Geometric Shapes

## DESCRIPTION

The `GXSetPaths` function copies the geometry information from the `data` parameter into the `geometry` property of the target path shape. If the target shape is not a path shape, this function posts the error code `illegal_type_for_shape`.

You must provide a pointer to a `gxPaths` structure in the `data` parameter—if you pass `nil` for this parameter, the function posts the error code `parameter_is_nil`.

If the target shape is locked (that is, its `gxLockedShape` shape attribute is set), this function posts the error `shape_access_not_allowed`.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>number_of_points_exceeds_implementation_limit</code>	
<code>number_of_contours_exceeds_implementation_limit</code>	
<code>size_of_path_exceeds_implementation_limit</code>	
<code>count_is_less_than_one</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)

## SEE ALSO

For general information about path geometries, see “Path Shapes” on page 2-25.

For the definition of the `gxPaths` structure, see page 2-107.

To create a new path shape, use the `GXNewPaths` function, which is described on page 2-117.

To examine the geometry of an existing path shape, use the `GXGetPaths` function, which is described on page 2-132.

To draw a path geometry without creating a path shape, use the `GXDrawPaths` function, which is described on page 2-162. To draw a path shape, use the `GXDrawShape` function, which is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## Editing Shape Geometries

---

The functions described in the previous section, “Getting and Setting Shape Geometries,” allow you to examine and replace entire shape geometries. The functions in this section provide more sophisticated abilities—with these functions, you can examine and edit specific parts of geometries.

For example, the `GXCountShapeContours` function allows you to determine the number of contours in a shape’s geometry. For polygon and path shapes, this number is an integral part of the geometry—it is the first value stored in the geometry; for other geometric shapes, this function simply returns 1.

Similarly, the `GXCountShapePoints` function returns the number of geometric points in a specified contour of a shape’s geometry.

The `GXGetShapeIndex` function returns the geometry index of a specific geometric point given a contour number and the index of the geometric point within the contour. (Remember, each geometric point in a geometry has an geometry index—if you consider a geometry as a list of geometric points starting from the first geometric point of the first contour to the last geometric point of the last contour, the geometry index of a particular geometric point is its position in this list.) You use geometry indexes to specify ranges of geometric points in many of the functions in this section.

You can use the `GXGetShapePoints` function to obtain a copy of a particular range of geometric points from a shape’s geometry, and you can use the `GXSetShapePoints` to replace a particular range of geometric points in a shape’s geometry.

You can use the `GXGetPolygonParts` function to extract a range of geometric points from an existing polygon shape and put them into a new polygon geometry. You can use the `GXSetPolygonParts` function to replace any range of geometric points in an existing polygon shape with any new polygon geometry.

Similarly, you can use the `GetPathsParts` function to extract a range of geometric points from an existing path shape and put them into a new path geometry, and you can use the `SetPathsParts` function to replace any range of geometric points in an existing path shape with any new path geometry.

The `GXGetShapeParts` and `GXSetShapeParts` functions allow the broadest editing control. With the `GXGetShapeParts` function, you can extract any range of geometric points from an existing shape and put them into a new shape. With the `GXSetShapeParts` function, you can replace any range of geometric points in an existing shape with the entire geometry of another shape.

## Geometric Shapes

You can apply `GXCountShapeContours`, `GXCountShapePoints`, `GXGetShapeIndex`, `GXGetShapePoints`, `GXSetShapePoints`, `GXGetShapeParts`, and `GXSetShapeParts` functions to other shape types as well as geometric shapes. Information about how they work for geometric shapes is presented in this section. You can find more information about these functions in Chapter 5, “Bitmap Shapes,” and Chapter 6, “Picture Shapes,” and in *Inside Macintosh: QuickDraw GX Typography*

## GXCountShapeContours

---

You can use the `GXCountShapeContours` function to determine the number of contours in a shape.

```
long GXCountShapeContours(gxShape source);
```

**source**      A reference to the shape whose contours you want to count.

**function result**   The number of contours in the source shape.

### DESCRIPTION

The `GXCountShapeContours` function returns as its function result the number of contours in the source shape. For polygon and path shapes, this number indicates the total number of polygon contours or path contours contained in the shape. For points, lines, curves, and rectangles, this function returns the value 1. For empty and full shapes, this function posts the `graphics_type_does_not_have_multiple_contours` error.

If you provide a source shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Always returns 1 as the function result
picture	Returns the number of picture items
text	Returns the number of glyphs
glyph	Returns the number of glyphs
layout	Returns the byte length of the text

## ERRORS, WARNINGS, AND NOTICES

**Errors**`out_of_memory``shape_is_nil``graphic_type_does_not_have_multiple_contours` (debugging version)

## SEE ALSO

For a discussion of contours, see “Shape Geometry” on page 2-9, “Polygon Shapes” on page 2-22, and “Path Shapes” on page 2-25.

To learn how this function works for typographic shape types, see *Inside Macintosh: QuickDraw GX Typography*.

To determine the number of points in a specific contour of a shape, use the `GXCountShapePoints` function, which is described in the next section.

## GXCountShapePoints

---

You can use the `GXCountShapePoints` function to determine the number of geometric points in a specific contour of a shape.

```
long GXCountShapePoints(gxShape source, long contour);
```

`source`        A reference to the shape containing the contour.

`contour`      The index of the contour whose geometric points you want to count.

*function result*   The number of points in the specified contour of the source shape.

## DESCRIPTION

The `GXCountShapePoints` function returns as its function result the number of points in the contour specified by the `contour` parameter of the shape specified by the `source` parameter. If you pass 0 for the `contour` parameter, this function returns the total number of geometric points in the shape.

## Geometric Shapes

For the geometric shapes with only one contour—points, lines, curves, and rectangles—you must pass a 0 or a 1 in the `contour` parameter. For polygons and paths shapes, the value you provide for the `contour` parameter must be 0 or greater and must be equal to or less than the actual number of contours in the shape. For empty and full shapes, the function posts a `contour_out_of_range` warning.

If you provide a source shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
<code>bitmap</code>	Returns 1 if the <code>contour</code> parameter is 0 or 1; posts the error <code>contour_out_of_range</code> otherwise
<code>picture</code>	Posts the error <code>graphic_type_does_not_contain_points</code>
<code>text</code>	Returns 1 if the <code>contour</code> parameter is 0 or 1; posts the error <code>contour_out_of_range</code> otherwise
<code>glyph</code>	Returns the number of glyphs in the style run indicated by the <code>contour</code> parameter
<code>layout</code>	Returns the byte length of the style run indicated by the <code>contour</code> parameter

## ERRORS, WARNINGS, AND NOTICES

**Errors**

`out_of_memory`  
`shape_is_nil`  
`graphic_type_does_not_contain_points` (debugging version)

**Warnings**

`contour_out_of_range`

## SEE ALSO

For a discussion of geometric points, see the section “About Geometric Shapes” beginning on page 2-5.

To learn how this function works for typographic shape types, see *Inside Macintosh: QuickDraw GX Typography*

To determine the number of contours in a shape, use the `GXCountShapeContours` function, which is described on page 2-136.

To determine the index of a particular geometric point within a shape, use the `GXGetShapeIndex` function, which is described in the next section.

## GXGetShapeIndex

---

You can use the `GXGetShapeIndex` function to determine the geometry index of a geometric point.

```
long GXGetShapeIndex(gxShape source, long contour, long vector);
```

<code>source</code>	A reference to the shape containing the desired geometric point.
<code>contour</code>	The index of the contour within the shape containing the geometric point.
<code>vector</code>	The index of the geometric point within that contour.

*function result* The geometry index of the specified geometric point.

### DESCRIPTION

The `GXGetShapeIndex` function returns as its function result the geometry index of the geometric point in the `source` shape's geometry that is identified by the `contour` and `vector` parameters. The indexes you provide in the `contour` and `vector` parameters are 1-based—for example, a value of 1 for the `contour` parameter indicates the first contour, and value of 2 indicates the second contour, and so on.

Each geometric point in a geometry has a geometry index—if you consider a geometry as a list of geometric points starting from the first geometric point of the first contour to the last geometric point of the last contour, the geometry index of a particular geometric point is its position in this list. For example, for a shape with two contours, the first with 10 geometric points and the second with 5 geometric points, this function would return 14 if you set the `contour` parameter to 2 and the `vector` parameter to 4.

For the geometric shapes with only one contour—points, lines, curves, and rectangles—you must pass a 1 in the `contour` parameter. For polygon and path shapes, the value you provide for the `contour` parameter must be greater than 0 and must be equal to or less than the actual number of contours in the shape. Otherwise, the function posts a `contour_out_of_range` warning. Similarly, the value you provide for the `vector` parameter must be equal to or less than the actual number of geometric points in the specified contour, or the function posts an `index_out_of_range_in_contour` warning and returns 0 as the function result.

If you provide a source shape that is an empty shape, a full shape, or a shape that is not one of the geometric shape types, this function posts the error `graphic_type_does_not_have_multiple_contours`.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>index_is_less_than_one</code>	(debugging version)
<code>graphic_type_does_not_contain_points</code>	(debugging version)
<code>graphic_type_does_not_have_multiple_contours</code>	(debugging version)

**Warnings**

<code>contour_out_of_range</code>
<code>index_out_of_range_in_contour</code>

## SEE ALSO

For a discussion of geometric points, see the section “About Geometric Shapes” beginning on page 2-5.

To determine the number of contours in a shape, use the `GXCountShapeContours` function, which is described on page 2-136.

To determine the number of geometric points in a contour, use the `GXCountShapePoints` function, which is described on page 2-137.

To copy a range of geometric points from a shape’s geometry, use the `GXGetShapePoints` function, which is described in the next section.

**GXGetShapePoints**

---

You can use the `GXGetShapePoints` function to obtain a copy of a range of geometric points from a specified shape.

```
long GXGetShapePoints(gxShape source, long index, long count,
                     gxPoint data[]);
```

<code>source</code>	A reference to the shape containing the desired geometric points.
<code>index</code>	The geometry index of the first geometric point to copy.
<code>count</code>	The number of the geometric points to copy. You may provide the <code>gxSelectToEnd</code> constant for this parameter.
<code>data</code>	A pointer to an array of <code>gxPoint</code> structures. On return, this array contains the copied points.

*function result* The number of geometric points copied.



**DESCRIPTION**

The `GXGetShapePoints` function returns in the `data` parameter a copy of the geometric points from the source shape's geometry starting from the geometric point with the geometry index indicated in the `index` parameter.

You provide, in the `count` parameter, the number of geometric points you want copied. The function result is the actual number of points copied. Typically, the value you provide for the `count` parameter is the same as the function result returned by this function. There are two exceptions:

- n If you provide too large a value for the `count` parameter—that is, the geometry of the source shape does not have enough geometric points to satisfy your request—this function copies as many geometric points as the shape does have (starting from the geometric point with the geometry index indicated by the `index` parameter). In this case, the function posts a `count_out_of_range` warning, and the function result reflects the actual number of geometric points copied.
- n Similarly, if you set the `count` parameter to the `gxSelectToEnd` constant, the function copies as many geometric points as the shape has, starting from the geometric point with the geometry index indicated by the `index` parameter. In this case, the function result reflects the actual number of geometric points copied, but no warning is posted.

Notice that this function returns the copied points as a single point array. If the source shape is a polygon or path shape, the information about which contours contained the geometric points is not retained.

If you want use the `gxSelectToEnd` constant for the `count` parameter, you would typically do the following:

1. Determine the length of the point array by calling this function, passing `nil` for the `data` parameter.
2. Allocate enough memory to hold the point array.
3. Call this function again, passing a pointer to the allocated memory in the `data` parameter.

If you provide an empty or full shape for the source shape, this function posts the error `graphic_type_does_not_contain_points`.

If you provide a source shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Always returns 1 as the function result
picture	Posts the error <code>graphic_type_does_not_contain_points</code>
text	Always returns 1
glyph	Returns the number of glyphs
layout	Always returns 1

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)
<code>graphic_type_does_not_contain_points</code>	(debugging version)

**Warnings**

<code>index_out_of_range_in_contour</code>
<code>count_out_of_range</code>

## SEE ALSO

For a discussion of geometric points, see the section “Shape Geometry” beginning on page 2-9.

To learn how this function works for typographic shape types, see *Inside Macintosh: QuickDraw GX Typography*.

To determine the geometry index of a particular geometric point within a shape’s geometry, use the `GXGetShapeIndex` function, which is described on page 2-139.

To replace a range of geometric points in a geometry, use the `GXSetShapePoints` function, which is described in the next section.

**GXSetShapePoints**

---

You can use the `GXSetShapePoints` procedure to replace geometric points of a shape.

```
void GXSetShapePoints(gxShape target, long index, long count,
                     const gxPoint data[]);
```

<code>target</code>	A reference to the shape containing the geometric points you want to replace.
<code>index</code>	The geometry index of the first geometric point to replace.
<code>count</code>	The number of the geometric points to replace.
<code>data</code>	An array of new geometric points.

## DESCRIPTION

The `GXSetShapePoints` function changes the values of the number of geometric points specified in the `count` parameter, starting with the geometric point indicated by the `index` parameter, to the values specified by the `data` parameter.

Notice that this function replaces geometric points on a point-by-point basis; the number of points in the `data` parameter must match the value of the `count` parameter. You may not use the `gxSelectToEnd` constant for the `count` parameter.

## Geometric Shapes

If you provide an empty or full shape for the source shape, this function posts the error `graphic_type_does_not_contain_points`.

If you provide a source shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Sets bitmap position
picture	Posts the error <code>graphic_type_does_not_contain_points</code>
text	Sets position of text shape
glyph	Sets glyph positions corresponding to range indicated by the <code>index</code> and <code>count</code> parameters
layout	Sets position of layout shape

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)
<code>graphic_type_does_not_contain_points</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)

**Warnings**

<code>index_out_of_range_in_contour</code>
<code>count_out_of_range</code>

## SEE ALSO

For examples that use this function, see “Replacing Geometric Points” beginning on page 2-79.

For a discussion of geometric points, see the section “Shape Geometry” beginning on page 2-9.

To learn how this function works for typographic shape types, see *Inside Macintosh: QuickDraw GX Typography*.

To determine the geometry index of a particular geometric point within a shape, use the `GXGetShapeIndex` function, which is described on page 2-139.

To obtain a copy of a range of geometric points in a geometry, use the `GXGetShapePoints` function, which is described on page 2-140.

## GXGetPolygonParts

---

You can use the `GXGetPolygonParts` function to copy a specified range of geometric points from the geometry of a polygon shape and then put these points into a polygon structure.

```
long GXGetPolygonParts(gxShape source, long index, long count,
                       gxPolygons *data);
```

<code>source</code>	A reference to the polygon shape containing the desired geometric points.
<code>index</code>	The geometry index of the first geometric point to copy.
<code>count</code>	The number of the geometric points to copy. You may provide the <code>gxSelectToEnd</code> constant for this parameter.
<code>data</code>	A pointer to a polygon structure to hold the copied geometric information.

*function result* The number of bytes required to hold the information returned in the `data` parameter.

### DESCRIPTION

The `GXGetPolygonParts` function copies geometry information from the source polygon shape into the polygon structure specified by the `data` parameter. This function copies all of the geometry information starting with the geometric point indicated by the `index` parameter and continuing for as many geometric points as indicated by the `count` parameter. This function copies the values of the indicated geometric points and retains the information about contour breaks from the original geometry. The function result is the length in bytes of the information returned in the `data` parameter.

Both the `index` and the `count` parameters must be greater than 0, although you can provide the `gxSelectToEnd` constant for the `count` parameter, which indicates that you want a copy of all the geometric points starting with the point indicated by the `index` parameter.

You may pass `nil` for the `data` parameter. In this case, the function still returns the byte length as the function result, but does not copy any geometry information.

Typically, to use this function, you go through these steps:

1. Determine the byte length needed to store the copied geometry information by calling this function, passing `nil` for the `data` parameter.
2. Allocate enough memory to hold the copied geometric information.
3. Call this function again, passing a pointer to the allocated memory in the `data` parameter.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>illegal_type_for_shape</code>	(debugging version)
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)

**Warnings**

<code>index_out_of_range</code>
<code>count_out_of_range</code>

## SEE ALSO

For an example that uses this function, see “Editing Polygon Parts” beginning on page 2-82.

For a discussion of polygons, see “Polygon Shapes” on page 2-22.

For the definition of the `gxPolygons` structure, see page 2-106.

For information about other functions that allow you to extract information from shape geometries, see the description of the `GXGetShapePoints` function on page 2-140 and the description of the `GXGetShapeParts` function on page 2-152.

To replace parts of a polygon shape’s geometry, use the `GXSetPolygonParts` function, which is described in the next section.

## GXSetPolygonParts

---

You can use the `GXSetPolygonParts` function to replace a range of geometry information in the geometry of a polygon shape with information from a specified polygon structure.

```
void GXSetPolygonParts(gxShape target, long index, long count,
                      const gxPolygons *data,
                      gxEditShapeFlag flags);
```

<code>target</code>	A reference to the polygon shape whose geometry you want to edit.
<code>index</code>	The geometry index of the first geometric point to replace. A value of 0 indicates that the new information should be inserted after the final geometric point in the target shape’s geometry.
<code>count</code>	The number of the geometric points to replace. A value of 0 indicates that no geometric points should be replaced; instead, the new information is inserted before the geometric point indicated by the <code>index</code> parameter. If you pass the <code>gxSelectToEnd</code> constant for this parameter, all geometric points starting with the geometric point indicated by the <code>index</code> parameter are replaced.

## Geometric Shapes

<code>data</code>	A pointer to a polygon structure containing the new geometry information.
<code>flags</code>	A set of flags that determine how the new information is inserted in the existing geometry.

## DESCRIPTION

The `GXSetPolygonParts` function replaces geometry information in the target shape's geometry with the information pointed to by the `data` parameter. The `index` and `count` parameters determine what part of the original geometry is replaced. The `flags` parameter determines how the new information is inserted in the geometry.

The `data` parameter contains a pointer to the geometry information to be copied into the target shape's geometry. If you pass the `gxSetToNil` constant for this parameter, no new information is copied in; in this case, this function removes the indicated geometric points instead of replacing them.

The `index` parameter indicates the first geometric point to be replaced. If you pass a value of 0 for this parameter, no geometric points are replaced. Instead, this function inserts the new geometry information after the last geometric point of the target shape's original geometry. If you pass 0 for this parameter, you must pass 0 or the `gxSelectToEnd` constant for the `count` parameter.

The `count` parameter indicates how many geometric points in the original geometry should be replaced. If you pass a value of 0 for this parameter, no geometric points are replaced; instead, this function inserts the new geometry information before the geometric point indicated by the `index` parameter. If you pass the `gxSelectToEnd` constant for this parameter, the function replaces all geometric points in the original geometry starting with the geometric point indicated by the `index` parameter.

When this function inserts the new geometry information, it retains the contour breaks contained in the `gxPolygons` structure specified by the `data` parameter. For example, if you provide a `gxPolygons` structure that contains two contours, the break between those contours remains when the new geometric points are inserted in the target shape's geometry.

The `flags` parameter indicates how you want the function to merge the first geometric point and the last geometric point of the `gxPolygons` structure into the target shape's geometry. The possible flags are

<code>gxBreakNeitherEdit</code>	= 0
<code>gxBreakLeftEdit</code>	= 0x01
<code>gxBreakRightEdit</code>	= 0x02
<code>gxRemoveDuplicatePoints</code>	= 0x04

## Geometric Shapes

The `gxBreakNeitherEdit` value indicates that the first geometric point of the `gxPolygons` structure should be merged into the preceding contour of the target shape's geometry and the final geometric point of the `gxPolygons` structure should be merged into the following contour.

The `gxBreakLeftEdit` flag indicates that the first geometric point of the `gxPolygons` structure should begin a new contour in the target shape's geometry. The `gxBreakRightEdit` flag indicates that the geometric point in the target shape that follows the final geometric point of the `gxPolygons` structure (after the new information is inserted) should begin a new contour.

The `gxRemoveDuplicatePoints` flag indicates that this function should, when inserting the information from the `gxPolygons` structure, remove the first geometric point of this structure if it exactly matches the preceding geometric point. Similarly, this flag indicates that the final geometric point of the `gxPolygons` structure should be removed if it exactly matches the subsequent geometric point in the target shape's geometry.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>number_of_points_exceeds_implementation_limit</code>	
<code>number_of_contours_exceeds_implementation_limit</code>	
<code>size_of_polygon_exceeds_implementation_limit</code>	
<code>illegal_type_for_shape</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>count_is_less_than_zero</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)

**Warnings**

<code>index_out_of_range</code>
<code>count_out_of_range</code>

## SEE ALSO

For an example that uses this function, see “Editing Polygon Parts” beginning on page 2-82.

For a discussion of polygons, see “Polygon Shapes” on page 2-22.

For the definition of the `gxPolygons` structure, see page 2-106.

For information about other functions that allow you to edit information in shape geometries, see the description of the `GXSetShapePoints` function on page 2-142 and the description of the `GXSetShapeParts` function on page 2-154.

To copy parts of a polygon shape's geometry, use the `GXGetPolygonParts` function, which is described on page 2-144.

## GXGetPathParts

---

You can use the `GXGetPathParts` function to extract a copy of a specified range of geometric points from the geometry of a path shape and put these points into a `gxPaths` structure.

```
long GXGetPathParts(gxShape source, long index, long count,
                    gxPaths *data);
```

<code>source</code>	A reference to the path shape containing the desired geometric points.
<code>index</code>	The geometry index of the first geometric point to copy.
<code>count</code>	The number of geometric points to copy. You may provide the <code>gxSelectToEnd</code> constant for this parameter.
<code>data</code>	A pointer to a <code>gxPaths</code> structure. On return, this structure contains the copied geometric information.

*function result* The number of bytes required to hold the information returned in the `data` parameter.

### DESCRIPTION

The `GXGetPathParts` function copies geometry information from the source path shape into the `gxPaths` structure specified by the `data` parameter. This function copies all of the geometry information starting with the geometric point indicated by the `index` parameter and continuing for as many geometric points as indicated by the `count` parameter. This function copies the values of the indicated geometric points and retains the information about contour breaks from the original geometry, as well as the information about which points are on curve and which are off curve. The function result is the length in bytes of the information returned in the `data` parameter.

Both the `index` and the `count` parameters must be greater than 0, although you can provide the `gxSelectToEnd` constant for the `count` parameter, which indicates that you want a copy of all the geometric points starting with the geometric point indicated by the `index` parameter.

You may pass `nil` for the `data` parameter. In this case, the function still returns the byte length as the function result, but does not copy any geometry information.

Typically, to use this function, you go through these steps:

1. Determine the byte length needed to store the copied geometry information by calling this function, passing `nil` for the `data` parameter.
2. Allocate enough memory to hold the copied geometry information.
3. Call this function again, passing a pointer to the allocated memory in the `data` parameter.



## ERRORS, WARNINGS, AND NOTICES

**Errors**

```

out_of_memory
shape_is_nil
illegal_type_for_shape      (debugging version)
index_is_less_than_one      (debugging version)
count_is_less_than_one      (debugging version)

```

**Warnings**

```

index_out_of_range
count_out_of_range

```

## SEE ALSO

For a discussion of paths, see “Path Shapes” on page 2-25.

For the definition of the `gxPaths` structure, see page 2-107.

For information about other functions that allow you to extract information from shape geometries, see the description of the `GXGetShapePoints` function on page 2-140 and the description of the `GXGetShapeParts` function on page 2-152.

To replace parts of a path shape’s geometry, use the `GXSetPathParts` function, which is described in the next section.

**GXSetPathParts**

---

You can use the `GXSetPathParts` function to replace a range of geometric points in the geometry of a path shape with the information from a specified `gxPaths` structure.

```

void GXSetPathParts(gxShape target, long index, long count,
                    const gxPaths *data, gxEditShapeFlag flags);

```

<code>target</code>	A reference to the path shape whose geometry you want to edit.
<code>index</code>	The index number of the first geometric point to replace. A value of 0 indicates that the new information should be inserted after the final geometric point in the target shape’s geometry.
<code>count</code>	The number of the geometric points to replace. A value of 0 indicates that no geometric points should be replaced; instead, the new information is inserted before the geometric point specified by the <code>index</code> parameter. If you pass the <code>gxSelectToEnd</code> constant for this parameter, all geometric points from the one specified by the <code>index</code> parameter to the final geometric point are replaced.
<code>data</code>	A pointer to the <code>gxPaths</code> structure containing the new geometry information.
<code>flags</code>	A set of flags that determine how the new information is inserted in the existing geometry.

## DESCRIPTION

The `GXSetPathParts` function replaces geometry information in the target shape's geometry with the information pointed to by the `data` parameter. The `index` and `count` parameters determine what part of the original geometry is replaced. The `flags` parameter determines how the new information is inserted in the geometry.

The `data` parameter contains a pointer to the geometry information to be copied into the target shape's geometry. If you pass the `gxSetToNil` constant for this parameter, no new information is copied in; in this case, the `GXSetPathParts` function removes the indicated geometric points instead of replacing them.

The `index` parameter indicates the first geometric point to be replaced. If you pass a value of 0 for this parameter, no geometric points are replaced. Instead, this function inserts the new geometric information after the last geometric point of the target shape's original geometry. If you pass 0 for this parameter, you must pass 0 or the `gxSelectToEnd` constant for the `count` parameter.

The `count` parameter indicates how many geometric points in the original geometry should be replaced. If you pass a value of 0 for this parameter, no geometric points are replaced; instead, this function inserts the new geometry information before the geometric point indicated by the `index` parameter. If you pass the `gxSelectToEnd` constant for this parameter, the function replaces all geometric points in the original geometry starting with the one indicated by the `index` parameter.

When this function inserts the new geometric information, it retains the contour breaks contained in the `gxPaths` structure specified in the `data` parameter. For example, if you provide a `gxPaths` structure that contains two contours, the break between those contours remains when the geometric points are inserted into the target shape's geometry.

The `flags` parameter indicates how you want the function to merge the first geometric point and the last geometric point of the `gxPaths` structure into the target shape's geometry. The possible flags are

<code>gxBreakNeitherEdit</code>	= 0
<code>gxBreakLeftEdit</code>	= 0x01
<code>gxBreakRightEdit</code>	= 0x02
<code>gxRemoveDuplicatePoints</code>	= 0x04

The `gxBreakNeitherEdit` value indicates that the first geometric point of the `gxPaths` structure should be merged into the preceding contour of the target shape's geometry and the final geometric point of the `gxPaths` structure should be merged into the subsequent contour.

## Geometric Shapes

The `gxBreakLeftEdit` flag indicates that the first geometric point of the `gxPaths` structure should begin a new contour once inserted in the target shape's geometry. The `gxBreakRightEdit` flag indicates that the geometric point in the target shape that follows the final geometric point of the `gxPaths` structure (after the new information is inserted) should begin a new contour.

The `gxRemoveDuplicatePoints` flag indicates that this function should, when inserting the information from the `gxPaths` structure, remove the first geometric point of this inserted structure if it exactly matches the preceding point in the existing geometry. Similarly, this flag indicates that the final geometric point of the `gxPaths` structure should be removed if it exactly matches the subsequent geometric point in the target shape's geometry.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>number_of_points_exceeds_implementation_limit</code>	
<code>number_of_contours_exceeds_implementation_limit</code>	
<code>size_of_polygon_exceeds_implementation_limit</code>	
<code>illegal_type_for_shape</code>	(debugging version)
<code>inconsistent_parameters</code>	(debugging version)
<code>index_is_less_than_zero</code>	(debugging version)
<code>count_is_less_than_zero</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)

**Warnings**

<code>index_out_of_range</code>
<code>count_out_of_range</code>

## SEE ALSO

For an example that uses this function, see “Editing Paths Parts” beginning on page 2-91.

For a discussion of paths, see “Path Shapes” on page 2-25.

For the definition of the `gxPaths` structure, see “Path Structures” on page 2-107.

For information about other functions that allow you to edit information in shape geometries, see the description of the `GXSetShapePoints` function on page 2-142 and the description of the `GXSetShapeParts` function on page 2-154.

To copy parts of a path shape's geometry, use the `GXGetPathParts` function, which is described on page 2-148.

## GXGetShapeParts

---

You can use the `GXGetShapeParts` function to extract a copy of a specified range of geometric points from the geometry of one shape and encapsulate it in another shape.

```
gxShape GXGetShapeParts(gxShape source, long index, long count,
                        gxShape destination);
```

<code>source</code>	A reference to the shape containing the desired geometric points.
<code>index</code>	The geometry index of the first geometric point to copy.
<code>count</code>	The number of geometric points to copy. You may provide the <code>gxSelectToEnd</code> constant for this parameter.
<code>destination</code>	A reference to the shape to encapsulate the copied geometry information.

*function result* A copy of the reference returned in the `destination` parameter.

### DESCRIPTION

The `GXGetShapeParts` function copies geometry information from the source shape into the destination shape. This function copies all of the geometry information starting with the geometric point indicated by the `index` parameter and continuing for as many geometric points as indicated by the `count` parameter. This function copies the values of the indicated geometric points and retains the information about contour breaks from the original geometry, as well as the information about which points are on curve and which are off curve. As a convenience, the function returns as its function result a reference to the destination shape.

Both the `index` and the `count` parameters must be greater than 0, although you can provide the `gxSelectToEnd` constant for the `count` parameter, which indicates that you want a copy of all the geometric points (starting with the geometric point indicated by the `index` parameter) in the source shape's geometry.

You may pass `nil` for the `destination` parameter. In this case, the function creates a new shape of the appropriate type and encapsulates the extracted geometry information in this new shape.

## Geometric Shapes

If the source shape is one of the geometric shape types, this function returns a geometric shape type, as described in the following table:

Shape type	Action taken
empty	Returns an empty shape
full	Returns a full shape
point	Returns a point shape
line	Returns a point or a line shape, depending on the number of geometric points copied
curve	Returns a point, line, or curve shape, depending on the number of geometric points copied
rectangle	Returns a point or a rectangle shape, depending on the number of geometric points copied
polygon	Always returns a polygon shape, even if only one or two geometric points are copied
path	Always returns a path shape, even if only one, two, or three geometric points are copied

If you provide a source shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts <code>shape_operator_may_not_be_a_bitmap</code> error
picture	Returns the number of picture items
text	Returns the number of glyphs
glyph	Returns the number of glyphs
layout	Returns the byte length of the text

## SPECIAL CONSIDERATIONS

If you pass `nil` for the `destination` parameter and no error results, the `GXGetShapeParts` function creates a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about creating and disposing of objects.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)

**Warnings**

<code>shape_operator_may_not_be_a_bitmap</code>
<code>index_out_of_range</code>
<code>count_out_of_range</code>

## SEE ALSO

For information about other functions that allow you to extract information from shape geometries, see the description of the `GXGetShapePoints` function on page 2-140, the description of the `GXGetPolygonParts` function on page 2-144, and the description of the `GXGetPathParts` function on page 2-148.

To replace parts of a shape's geometry, use the `GXSetShapeParts` function, which is described in the next section.

**GXSetShapeParts**

---

You can use the `GXSetShapeParts` function to replace a range of geometric points in a shape's geometry with the information in another shape's geometry.

```
void GXSetShapeParts(gxShape target, long index, long count,
                    gxShape insert, gxEditShapeFlag flags);
```

<code>target</code>	A reference to the shape whose geometry you want to edit.
<code>index</code>	The geometry index of the first geometric point to replace. A value of 0 indicates that the new information should be inserted after the final geometric point in the target shape's geometry.
<code>count</code>	How many geometric points to replace. A value of 0 indicates that no geometric points should be replaced; instead, the new information is inserted before the geometric point specified by the <code>index</code> parameter. If you pass the <code>gxSelectToEnd</code> constant for this parameter, all geometric points from the one specified by the <code>index</code> parameter to the final one are replaced.
<code>insert</code>	A reference to the shape whose geometry you want to insert. You may specify the <code>gxSetToNil</code> constant for this parameter to indicate that you want to delete points from the target shape's geometry.
<code>flags</code>	A set of flags that determine how the new geometry information is inserted in the <code>target</code> shape's geometry.

## DESCRIPTION

The `GXSetShapeParts` function replaces geometry information in the target shape's geometry with the geometry information in the shape specified by the `insert` parameter. The `index` and `count` parameters determine what part of the original geometry is replaced. The `flags` parameter determines how the new information is inserted in the geometry.

This function converts the shape type of the target shape to be suitable to hold the information from the inserted shape. For example, if the target shape is a line and the inserted shape is a rectangle, this function converts the target shape to a polygon shape before inserting the rectangle.

If the target shape is a rectangle, you may only insert information before both geometric points, after both geometric points, or in place of both geometric points.

You may add any shape to an empty target shape—the result will be identical to the inserted shape. You may also add any shape to a full target shape, but the result will also be a full shape.

The `index` parameter indicates the first geometric point to be replaced. If you pass a value of 0 for this parameter, no geometric points are replaced. Instead, this function inserts the new geometry information after the last geometric point of the target shape's original geometry. If you pass a 0 for this parameter, you must pass a 0 or the `gxSelectToEnd` constant for the `count` parameter.

The `count` parameter indicates how many geometric points in the original geometry should be replaced. If you pass a value of 0 for this parameter, no geometric points are replaced; instead, this function inserts the new geometry information before the geometric point indicated by the `index` parameter. If you pass the `gxSelectToEnd` constant for this parameter, the function replaces all geometric points in the original geometry starting with the geometric point indicated by the `index` parameter.

When this function inserts the new geometry information, it retains the contour breaks contained in the inserted shape's geometry. For example, if you provide a path shape for the inserted shape that contains two contours, the break between those contours remains when the geometric points are inserted into the target shape's geometry.

The `flags` parameter indicates how you want the function to merge the first geometric point and the last geometric point of the inserted shape's geometry into the target shape's geometry. The possible flags are:

<code>gxBreakNeitherEdit</code>	= 0
<code>gxBreakLeftEdit</code>	= 0x01
<code>gxBreakRightEdit</code>	= 0x02
<code>gxRemoveDuplicatePoints</code>	= 0x04

The `gxBreakNeitherEdit` value indicates that the first geometric point of the inserted shape's geometry should be merged into the preceding contour of the target shape's geometry and the final geometric point of the inserted shape's geometry should be merged into the subsequent contour.

## Geometric Shapes

The `gxBreakLeftEdit` flag indicates that the first geometric point of the inserted shape's geometry should begin a new contour once inserted in the `target` shape's geometry. The `gxBreakRightEdit` flag indicates that the geometric point in the `target` shape that follows the final geometric point of the inserted shape's geometry (after the new information is inserted) should begin a new contour.

The `gxRemoveDuplicatePoints` flag indicates that this function should, when inserting the information from the inserted shape's geometry, remove the first geometric point of this inserted geometry if it exactly matches the preceding point in the existing geometry. Similarly, this flag indicates that the final geometric point of the inserted shape's geometry should be removed if it exactly matches the subsequent geometric point in the `target` shape's geometry.

If you provide a source shape that is a full shape, this function returns a full shape in the `destination` parameter.

If you provide a source shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
<code>bitmap</code>	Posts the error <code>shape_operator_may_not_be_a_bitmap</code>
<code>picture</code>	Calls the <code>GXSetPictureParts</code> function
<code>text</code>	Calls the <code>GXSetTextParts</code> function
<code>glyph</code>	Calls the <code>GXSetGlyphParts</code> function
<code>layout</code>	Calls the <code>GXSetLayoutParts</code> function

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)
<code>functionality_unimplemented</code>	(debugging version)
<code>rectangles_cannot_be_inserted_into</code>	(debugging version)
<code>shape_operator_may_not_be_a_bitmap</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)

**Warnings**

<code>index_out_of_range</code>
<code>count_out_of_range</code>
<code>picture_cannot_contain_itself</code>

**Notices**

<code>parameters_have_no_effect</code>	(debugging version)
--	---------------------



## SEE ALSO

For an example of this function, see “Editing Shape Parts” beginning on page 2-93.

To learn how this function works for typographic shape types, see *Inside Macintosh: QuickDraw GX Typography*.

For information about other functions that allow you to edit information in shape geometries, see the description of the `GXSetShapePoints` function on page 2-142, the description of the `GXSetPolygonParts` function on page 2-145, and the description of the `GXSetPathParts` function on page 2-149.

To copy parts of a shape’s geometry, use the `GXGetShapeParts` function, which is described on page 2-152.

## Drawing Geometric Shapes

---

The QuickDraw GX drawing functions compile all of the information in a shape’s properties, and the properties of its style, ink, and transform objects, and produce a graphic image. Therefore, to understand how these functions draw geometric shapes, you need to be familiar with much of the information in *Inside Macintosh: QuickDraw GX Objects*, as well as much of the information in this chapter and in the next chapter, “Geometric Styles.” The function descriptions in this section give an overview of the process these functions use to draw geometric shapes.

If you want to draw a geometric shape without creating a shape object—that is, just given a geometry—you can use the `GXDrawPoint`, `GXDrawLine`, `GXDrawCurve`, `GXDrawRectangle`, `GXDrawPolygons`, or `GXDrawPaths` functions, which are described in this section. These functions create a shape object, initialize it, draw it, and dispose of it; therefore, they do not take advantage of the QuickDraw GX caching mechanism. You should make limited use of these functions—for example, you could use one of these functions if you wanted to draw a particular shape drawn only once.

To draw a shape once you have created a shape object and modified its properties to suit your needs, you can use the `GXDrawShape` function. This function draws all shape types, and is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

When debugging your application, you can use the `GXGetDrawError` function, which is described in the chapter “QuickDraw GX Debugging” in *Inside Macintosh: QuickDraw GX Environment and Utilities*, for hints when a shape fails to draw as expected.

## GXDrawPoint

---

You can use the `GXDrawPoint` function to draw a point without creating a point shape.

```
void GXDrawPoint(const gxPoint *data);
```

`data`                    A pointer to the point geometry you want to draw.

### DESCRIPTION

The `GXDrawPoint` function draws the point geometry specified by the `data` parameter, using the shape fill, style, ink, and transform of the default point shape.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`  
`parameter_is_nil`            (debugging version)

### SEE ALSO

For examples using this function, see “Creating and Drawing Points” beginning on page 2-29.

For more information about points and the default point shape, see “Point Shapes” on page 2-16.

For the definition of the `gxPoint` structure, see page 2-104.

For more information about drawing shapes, see the description of the `GXDrawShape` function in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXDrawLine

---

You can use the `GXDrawLine` function to draw a line without creating a line shape.

```
void GXDrawLine(const gxLine *data);
```

`data`                    A pointer to the line geometry you want to draw.

### DESCRIPTION

The `GXDrawLine` function draws the line geometry specified by the `data` parameter, using the shape fill, style, ink, and transform of the default line shape.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

out\_of\_memory  
parameter\_is\_nil (debugging version)

## SEE ALSO

For examples using this function, see “Creating and Drawing Lines” beginning on page 2-36.

For more information about lines and the default line shape, see “Line Shapes” on page 2-17.

For the definition of the `gxLine` structure, see page 2-105.

For more information about drawing shapes, see the description of the `GXDrawShape` function in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

**GXDrawCurve**

---

You can use the `GXDrawCurve` function to draw a curve without creating a curve shape.

```
void GXDrawCurve(const gxCurve *data);
```

`data`            A pointer to the curve geometry you want to draw.

## DESCRIPTION

The `GXDrawCurve` function draws the curve geometry specified by the `data` parameter, using the shape fill, style, ink, and transform of the default curve shape.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

out\_of\_memory  
parameter\_is\_nil (debugging version)

## SEE ALSO

For examples using this function, see “Creating and Drawing Curves” beginning on page 2-41.

For more information about curves and the default curve shape, see “Curve Shapes” on page 2-18.

For the definition of the `gxCurve` structure, see page 2-105.

For more information about drawing shapes, see the description of the `GXDrawShape` function in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXDrawRectangle

---

You can use the `GXDrawRectangle` function to draw a rectangle without creating a rectangle shape.

```
void GXDrawRectangle(const gxRectangle *data, gxShapeFill fill);
```

`data`            A pointer to the rectangle geometry you want to draw.

`fill`            The shape fill to use when drawing the rectangle.

### DESCRIPTION

The `GXDrawRectangle` function draws the rectangle geometry specified by the `data` parameter, using the shape fill specified by the `fill` parameter, and the style, ink, and transform of the default rectangle shape.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`

`parameter_is_nil`

`shapeFill_is_not_allowed`

(debugging version)

(debugging version)

### SEE ALSO

For examples using this function, see “Creating and Drawing Rectangles” beginning on page 2-43.

For more information about rectangles and the default rectangle shape, see “Rectangle Shapes” on page 2-20.

For the definition of the `gxRectangle` structure, see page 2-106.

For more information about drawing shapes, see the description of the `GXDrawShape` function in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXDrawPolygons

---

You can use the `GXDrawPolygons` function to draw polygon contours without creating a polygon shape.

```
void GXDrawPolygons(const gxPolygons *data, gxShapeFill fill);
```

`data`            A pointer to the polygon geometry you want to draw.

`fill`            The shape fill to use when drawing the polygon contours.

### DESCRIPTION

The `GXDrawPolygons` function draws the polygon geometry specified by the `data` parameter, using the shape fill specified by the `fill` parameter, and the style, ink, and transform of the default polygon shape.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`  
`parameter_is_nil`      (debugging version)

### SEE ALSO

For more information about polygons and the default polygon shape, see “Polygon Shapes” on page 2-22.

For the definition of the `gxPolygons` structure, see page 2-106.

For examples using this function, see “Creating and Drawing Polygons” beginning on page 2-45.

For more information about drawing shapes, see the description of the `GXDrawShape` function in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXDrawPaths

---

You can use the `GXDrawPaths` function to draw path contours without creating a path shape.

```
void GXDrawPaths(const gxPaths *data, gxShapeFill fill);
```

`data`            A pointer to the path geometry you want to draw.

`fill`            The shape fill to use when drawing the path contours.

### DESCRIPTION

The `GXDrawPaths` function draws the path geometry specified by the `data` parameter, using the shape fill specified by the `fill` parameter, and the style, ink, and transform of the default path shape.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`  
`parameter_is_nil`      (debugging version)

### SEE ALSO

For more information about paths and the default path shape, see “Path Shapes” on page 2-25.

For the definition of the `gxPaths` structure, see page 2-107.

For examples using this function, see “Creating and Drawing Paths” beginning on page 2-55.

For more information about drawing shapes, see the description of the `GXDrawShape` function in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## Summary of Geometric Shapes

---

### Constants and Data Types

---

#### The Point Structure

```
struct gxPoint {  
    Fixed    x;  
    Fixed    y;  
};
```

#### The Line Structure

```
struct gxLine {  
    struct gxPoint first;  
    struct gxPoint last;  
};
```

#### The Curve Structure

```
struct gxCurve {  
    struct gxPoint first;  
    struct gxPoint control;  
    struct gxPoint last;  
};
```

#### The Rectangle Structure

```
struct gxRectangle {  
    Fixed    left;  
    Fixed    top;  
    Fixed    right;  
    Fixed    bottom;  
};
```

**Polygon Structures**

```

struct gxPolygon {
    long          vectors;
    struct gxPoint vector[gxAnyNumber];
};

struct gxPolygons {
    long          contours;
    struct gxPolygon contour[gxAnyNumber];
};

```

**Path Structures**

```

struct gxPath {
    long          vectors;
    long          controlBits[gxAnyNumber];
    struct gxPoint vector[gxAnyNumber];
};

struct gxPaths {
    long          contours;
    struct gxPath contour[gxAnyNumber];
};

```

**Functions**

---

**Creating Geometric Shapes**

```

gxShape GXNewPoint          (const gxPoint *data);
gxShape GXNewLine           (const gxLine *data);
gxShape GXNewCurve          (const gxCurve *data);
gxShape GXNewRectangle      (const gxRectangle *data);
gxShape GXNewPolygons       (const gxPolygons *data);
gxShape GXNewPaths          (const gxPaths *data);

```



**Getting and Setting Shape Geometries**

```

gxPoint *GXGetPoint          (gxShape source, gxPoint *data);
void GXSetPoint              (gxShape target, const gxPoint *data);
gxLine *GXGetLine           (gxShape source, gxLine *data);
void GXSetLine               (gxShape target, const gxLine *data);
gxCurve *GXGetCurve         (gxShape source, gxCurve *data);
void GXSetCurve              (gxShape target, const gxCurve *data);
gxRectangle *GXGetRectangle  (gxShape source, gxRectangle *data);
void GXSetRectangle          (gxShape target, const gxRectangle *data);
long GXGetPolygons           (gxShape source, gxPolygons *data);
void GXSetPolygons           (gxShape target, const gxPolygons *data);
long GXGetPaths              (gxShape source, gxPaths *data);
void GXSetPaths              (gxShape target, const gxPaths *data);

```

**Editing Shape Geometries**

```

long GXCountShapeContours    (gxShape source);
long GXCountShapePoints      (gxShape source, long contour);
long GXGetShapeIndex         (gxShape source, long contour, long vector);
long GXGetShapePoints        (gxShape source, long index, long count,
                             gxPoint data[]);
void GXSetShapePoints        (gxShape target, long index, long count,
                             const gxPoint data[]);
long GXGetPolygonParts       (gxShape source, long index, long count,
                             gxPolygons *data);
void GXSetPolygonParts       (gxShape target, long index, long count,
                             const gxPolygons *data,
                             gxEditShapeFlag flags);
long GXGetPathParts          (gxShape source, long index, long count,
                             gxPaths *data);
void GXSetPathParts          (gxShape target, long index, long count,
                             const gxPaths *data, gxEditShapeFlag flags);
gxShape GXGetShapeParts      (gxShape source, long index, long count,
                             gxShape destination);
void GXSetShapeParts         (gxShape target, long index, long count,
                             gxShape insert, gxEditShapeFlag flags);

```

**Drawing Geometric Shapes**

```
void GXDrawPoint          (const gxPoint *data);  
void GXDrawLine           (const gxLine *data);  
void GXDrawCurve          (const gxCurve *data);  
void GXDrawRectangle      (const gxRectangle *data, gxShapeFill fill);  
void GXDrawPolygons       (const gxPolygons *data, gxShapeFill fill);  
void GXDrawPaths          (const gxPaths *data, gxShapeFill fill);
```

# Geometric Styles

---

## Contents

About Geometric Styles	3-5
Shapes and Styles	3-5
Incorporating Stylistic Variations Into Shape Geometries	3-8
Style Properties	3-11
Default Style Objects	3-12
Curve Error	3-14
The Geometric Pen	3-15
Style Attributes	3-17
Pen Placement	3-18
Grids	3-20
Interactions Between Caps, Joins, Dashes, and Patterns	3-22
Caps	3-23
Joins	3-25
Dashes	3-27
Patterns	3-31
Interactions Between Caps, Joins, Dashes, and Patterns	3-33
Using Geometric Styles	3-35
Associating Styles With Shapes	3-36
Constraining Shape Geometries to Grids	3-40
Constraining Shapes to Device Grids	3-42
Using Curve Error When Converting Paths to Polygons	3-45
Using Curve Error When Reducing Shapes	3-49
Manipulating Pen Width and Placement	3-51
Adding Caps to a Shape	3-57
Adding Standard Caps to a Shape	3-59
Adding Joins to a Shape	3-61
Adding Standard Joins to a Shape	3-64
Dashing a Shape	3-66
Adjusting Dashes to Fit Contours	3-70
Insetting Dashes	3-73

Breaking and Bending Dashes	3-74
Wrapping Text to a Contour	3-80
Determining Dash Positions	3-81
Adding a Pattern to a Shape	3-86
Determining Pattern Positions	3-88
Combining Caps, Joins, Dashes, and Patterns	3-91
Geometric Styles Reference	3-96
Constants and Data Types	3-96
Style Objects	3-97
Style Attributes	3-98
The Cap Structure	3-99
Cap Attributes	3-101
The Join Structure	3-101
Join Attributes	3-102
The Dash Structure	3-103
Dash Attributes	3-105
The Pattern Structure	3-106
Pattern Attributes	3-107
Functions	3-108
Getting and Setting Style Attributes	3-109
GXGetStyleAttributes	3-109
GXSetStyleAttributes	3-110
GXGetShapeStyleAttributes	3-112
GXSetShapeStyleAttributes	3-113
Getting and Setting Curve Error	3-114
GXGetStyleCurveError	3-115
GXSetStyleCurveError	3-116
GXGetShapeCurveError	3-117
GXSetShapeCurveError	3-118
Getting and Setting the Pen Width	3-119
GXGetStylePen	3-119
GXSetStylePen	3-120
GXGetShapePen	3-121
GXSetShapePen	3-122
Getting and Setting Caps	3-123
GXGetStyleCap	3-124
GXSetStyleCap	3-125
GXGetShapeCap	3-126
GXSetShapeCap	3-128
Getting and Setting Joins	3-129
GXGetStyleJoin	3-129
GXSetStyleJoin	3-130
GXGetShapeJoin	3-132
GXSetShapeJoin	3-133
Getting and Setting Dashes	3-134
GXGetStyleDash	3-135
GXSetStyleDash	3-136

## CHAPTER 3

GXGetShapeDash	3-138
GXSetShapeDash	3-139
GXGetShapeDashPositions	3-140
<b>Getting and Setting Patterns</b>	<b>3-142</b>
GXGetStylePattern	3-142
GXSetStylePattern	3-144
GXGetShapePattern	3-145
GXSetShapePattern	3-146
GXGetShapePatternPositions	3-147
<b>Summary of Geometric Styles</b>	<b>3-149</b>
<b>Constants and Data Types</b>	<b>3-149</b>
<b>Functions for Manipulating Geometric Style Properties</b>	<b>3-151</b>



This chapter describes the geometric properties of style objects, which you can use to apply certain types of stylistic variations to QuickDraw GX shapes. In particular, this chapter shows how you can

- n constrain the drawing of a shape to a grid
- n specify the pen width to use when drawing a shape's frame
- n indicate the placement of the pen relative to the shape's frame
- n specify what to draw at the beginnings and the ends of a shape's contours
- n specify what to draw at the corners of a shape's contours
- n dash the contours of a shape
- n fill a shape, or the frame of a shape, with a pattern

You can also apply stylistic variations to typographic shapes, using the typographic properties of the style object. For example, you can use the style associated with a text shape to specify the text's font, font size, and type style. The chapter "Typographic Styles" in *Inside Macintosh: QuickDraw GX Typography* discusses the text-related properties of style objects.

You should be familiar with some of the information in *Inside Macintosh: QuickDraw GX Objects* before you read this chapter; in particular, you should read the chapters "Introduction to QuickDraw GX Objects" and "Style Objects" in that book.

## About Geometric Styles

---

A style is a group of stylistic variations applied to a shape. QuickDraw GX provides two major categories of stylistic variations: geometric variations, which include pen width, dashes, patterns, and so on, and typographic variations, which include font, font size, typestyle, and so on.

Both types of stylistic variation are encapsulated in a **style object**. Like a shape object, a style object is a data structure that you manipulate with functions provided by QuickDraw GX. Each style object has a group of properties, and each **style property** represents a different stylistic variation.

### Shapes and Styles

---

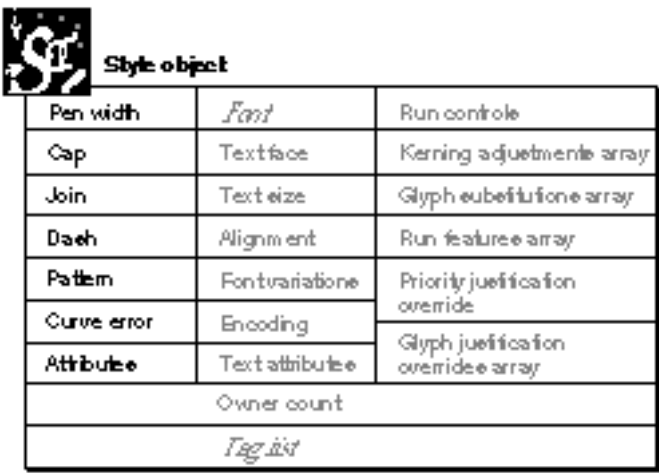
In general, a shape object is an object with a group of properties that describe a geometry; a style object is an object with a group of properties that affect how QuickDraw GX interprets a shape's geometry during drawing.

## Geometric Styles

Every QuickDraw GX shape object contains a reference to a style object. Figure 3-1 shows the properties of a style object. In this figure, the geometric properties—those that apply primarily to geometric shapes—are highlighted. These properties are discussed in this chapter. The other style properties are shown in gray. These include

- n the typographic style properties (those that apply primarily to typographic shapes), which are described in the chapter “Typographic Styles” in *Inside Macintosh: QuickDraw GX Typography*
- n the style properties common to all objects, (owner count and tag list), which are described in the chapter “Style Objects” in *Inside Macintosh: QuickDraw GX Objects*

**Figure 3-1** Style object with geometric properties highlighted

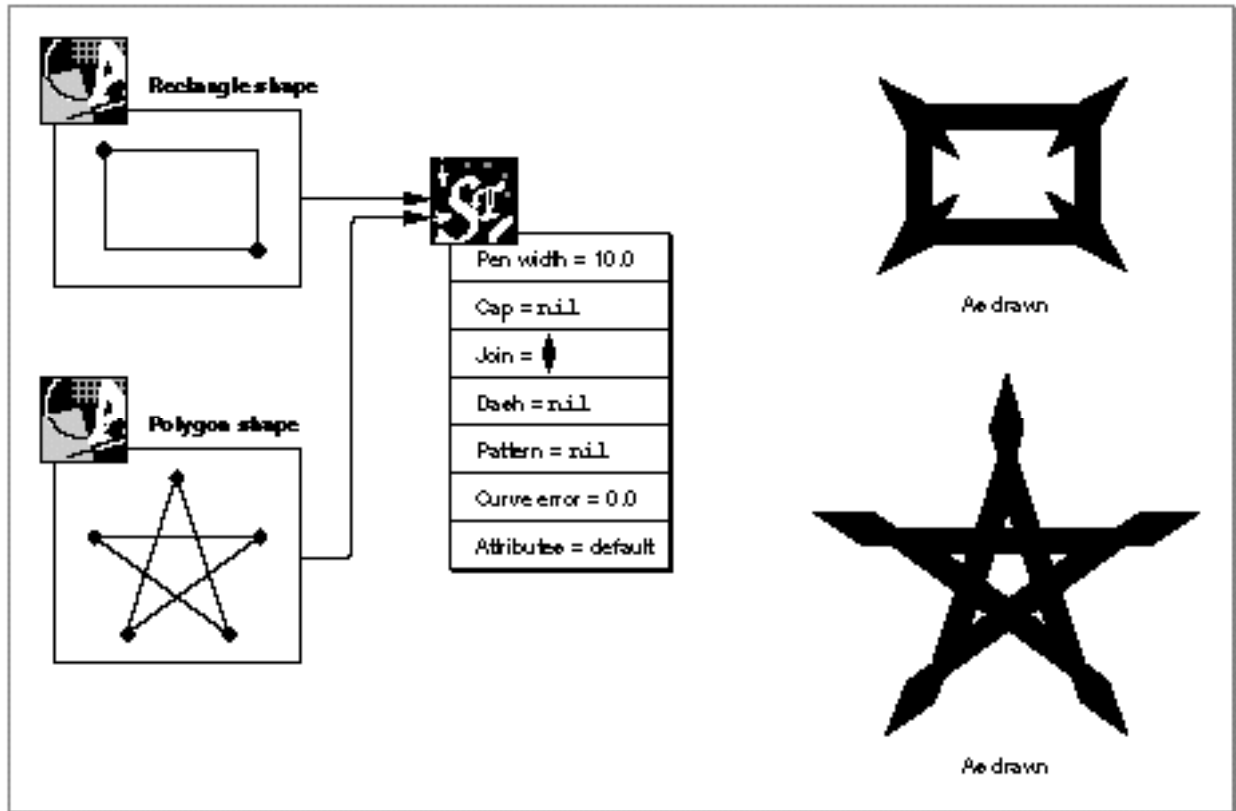


Property	Data Type	Data Structure
Pen width	<i>Font</i>	Run controls
Cap	Text face	Kerning adjustments array
Join	Text size	Glyph substitution array
Dash	Alignment	Run features array
Pattern	Font variations	Priority justification override
Curve error	Encoding	Glyph justification override array
Attribute	Text attributes	
Owner count		
<i>Tag list</i>		



As Figure 3-2 depicts, a single style object can be shared by multiple shape objects.

**Figure 3-2** Shared style objects



A geometric shape and a typographic shape can reference the same style object. The geometric shape uses the geometric style properties, which are described in this chapter, while the typographic shape uses the typographic style properties, which are described in the chapter “Typographic Styles” in *Inside Macintosh: QuickDraw GX Typography*.

QuickDraw GX typically handles style sharing for you. The section “Default Style Objects” on page 3-12 and the section “Associating Styles With Shapes” on page 3-36 describe the default style sharing behavior implemented by QuickDraw GX and how you can override this behavior.

As with all QuickDraw GX objects, a style object has an owner count, which reflects the number of existing references to the style object. When a new reference to a style object is created, the owner count of the style object is incremented; when a reference to a style object goes away, the owner count of the style object is decremented. When a style object has an owner count of 0, QuickDraw GX can free the memory used by the style object.

References to style objects typically include those contained in shape objects and those contained in variables in your application. QuickDraw GX manages the owner counts corresponding to references in shape objects for you; you are responsible for managing the owner counts corresponding to variables in your application. The chapter “Introduction to QuickDraw GX Objects” in *Inside Macintosh: QuickDraw GX Objects* explains owner counts and owner count management in more detail.

## Incorporating Stylistic Variations Into Shape Geometries

---

When you draw a shape, QuickDraw GX applies the information in the style object of the shape to the shape’s geometry. For example, style objects contain a pen width property, described in full later in this chapter. When you draw a line shape, QuickDraw GX draws the line with the width specified in the pen width property of the style object associated with the line shape. As drawn, the thick line looks like a filled polygon. However, even after drawing the line shape, the shape still contains a line geometry.

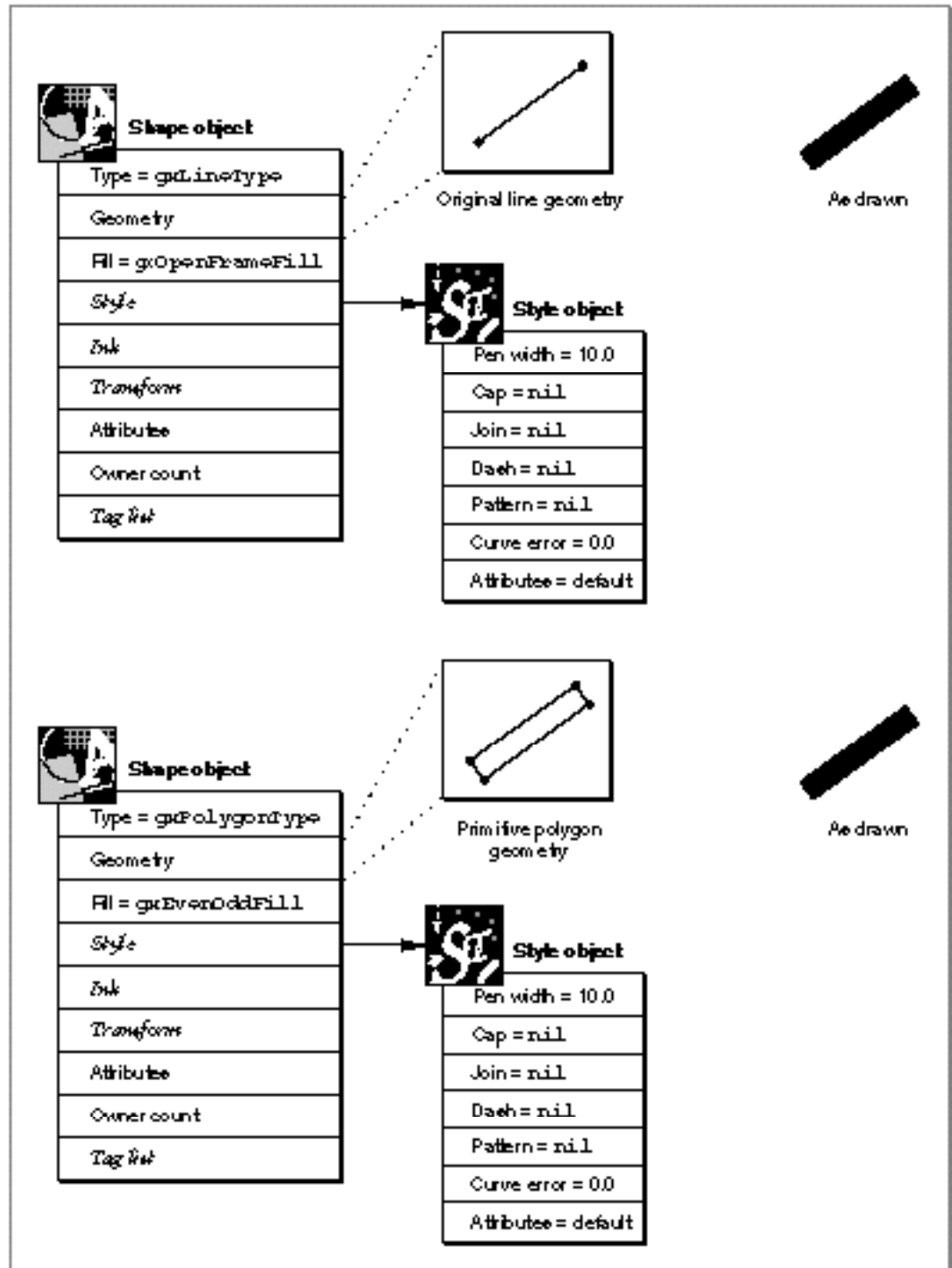
QuickDraw GX provides a mechanism for incorporating the stylistic variations contained in a style object directly into the geometry of a shape object. This mechanism is the `GXPrimitiveShape` function, which is described in full in the next chapter, “Geometric Operations.”

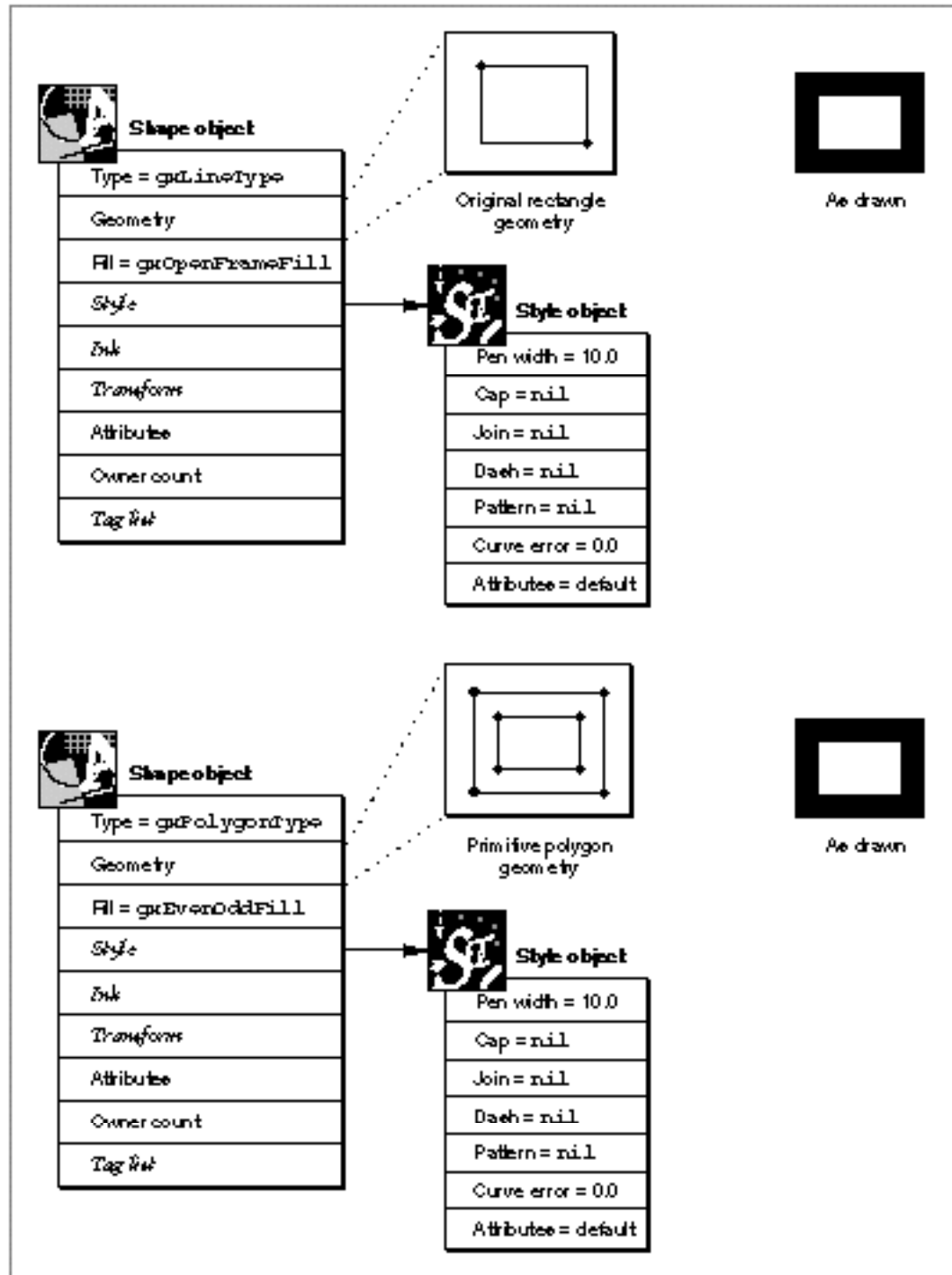
If you make changes to a shape’s style object and then call the `GXPrimitiveShape` function, QuickDraw GX changes the shape’s shape type, shape fill, and shape geometry to incorporate the new stylistic variations. Basically, the same process that happens when drawing the shape happens directly to the shape’s geometry.

For example, Figure 3-3 shows a line shape. If you alter the style of this line shape to include a pen width of 10, the line shape effectively becomes a filled polygon shape.

If you were to apply the `GXPrimitiveShape` function to this thick line shape, the `GXPrimitiveShape` function would change the shape type to the polygon type, the shape fill to even-odd shape fill, and the shape geometry to a list of the four geometric points that define the polygon, as shown in Figure 3-3.

Another example, the result of applying the `GXPrimitiveShape` function to a framed rectangle with a thick pen width is shown in Figure 3-4. In this case, the result of the `GXPrimitiveShape` function is a filled polygon shape with two contours: an inside contour and an outside contour.

**Figure 3-3** Effects of the GXPimitiveShape function on a line shape

**Figure 3-4** Effects of the `GXPrimitiveShape` function on a rectangle shape

## Geometric Styles

Notice that the `GXPrimitiveShape` function does not affect the style object of the shape: it merely incorporates the existing style information into the geometry of the shape.

The result of calling the `GXPrimitiveShape` function is called a **primitive shape**, or a shape in its primitive form. Primitive shapes include

- n empty shapes and full shapes, which are described in Chapter 3, “Geometric Shapes”
- n filled rectangle, polygon, and path shapes, which are also described in Chapter 3, “Geometric Shapes”
- n hairline framed shapes, which are described on page 3-16
- n glyph shapes, which are described in *Inside Macintosh: QuickDraw GX Typography*
- n bitmap shapes, which are described in Chapter 5, “Bitmap Shapes”

QuickDraw GX uses primitive shapes for caps, joins, dashes, and patterns, which are discussed throughout the rest of this chapter, and for clip shapes, which are discussed in the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## Style Properties

---

Like most QuickDraw GX objects, each style object has an owner count and a list of tags. These properties are described in detail in the chapter “Introduction to QuickDraw GX Objects” in *Inside Macintosh: QuickDraw GX Objects*.

In addition to the owner count and the tag list, each style object contains properties that primarily affect the drawing of geometric shapes and properties that primarily affect the drawing of typographic shapes.

The style properties that primarily affect geometric shapes include the following:

- n **Curve error.** This property specifies the allowable amount of error when QuickDraw GX converts a path shape into a polygon shape. It also specifies how far apart geometric points must be for QuickDraw GX to consider them separate points when simplifying or reducing a shape.
- n **Pen width.** This property specifies the width of the pen QuickDraw GX uses to draw the contours of a shape.
- n **Style attributes.** This property is a group of flags that allow you to specify how QuickDraw GX places the pen with respect to a shape’s geometry and whether the shape should be constrained to a grid when drawn.
- n **Caps.** This property specifies what QuickDraw GX should draw at the start and the end of a shape’s contours. QuickDraw GX allows you to use any geometric shape (for example, a polygon shaped like an arrow head) as a start cap or end cap.
- n **Join.** This property specifies what QuickDraw GX should draw at the corners of a shape’s contours. QuickDraw GX provides two standard join types (one for round corners and one for sharp corners), although QuickDraw GX allows you to specify any geometric shape as a join.

## Geometric Styles

- n **Dash.** This property specifies how QuickDraw GX should dash the contours of a shape. As with caps and joins, you can specify any geometric shape to dash the frame another shape. However, you can also dash a shape with glyphs, which gives the effect of fitting text to a shape's frame.
- n **Pattern.** This property specifies how QuickDraw GX should fill the geometry of a shape. You can use geometric shapes, glyphs shapes, or bitmap shapes as patterns.

The sections “Curve Error” on page 3-14, “The Geometric Pen” on page 3-15, “Style Attributes” on page 3-17, and “Interactions Between Caps, Joins, Dashes, and Patterns” on page 3-22 discuss these style properties in more detail.

The typographic style properties, which include font, text size, text face, and so on, are described in *Inside Macintosh: QuickDraw GX Typography*.

## Default Style Objects

---

When you call the `GXNewStyle` function, which is described in the chapter “Style Objects” in *Inside Macintosh: QuickDraw GX Objects*, QuickDraw GX creates and returns a new style object. All of the new style object's properties are set to standard initial values. Once you have created a new style object, you can change the values of its properties, but you cannot change the behavior of the `GXNewStyle` function itself; it always returns a style object with these values for the geometric style properties:

- n owner count: 1
- n tag list: no tags
- n style attributes: no attributes
- n curve error: 0.0
- n pen width: 0.0
- n cap
  - n cap attributes: no attributes
  - n start cap: none
  - n end cap: none
- n join
  - n join attributes: no attributes
  - n join: none
  - n join miter: Fixed1

## Geometric Styles

- n dash
  - n dash attributes: no attributes
  - n dash: none
  - n dash advance: 0.0
  - n dash phase: 0.0
  - n dash scale: Fixed1
- n pattern
  - n pattern attributes: no attributes
  - n pattern: none
  - n pattern grid: (0.0,0.0), (0.0,0.0)

The chapter “Typographic Styles” in *Inside Macintosh: QuickDraw GX Typography* discusses the default style values for the typographic style properties.

Although you cannot change the behavior of the `GXNewStyle` function, QuickDraw GX provides another method for creating new style objects—a method that you can modify. When you create a new shape with the `GXNewShape` function, QuickDraw GX returns a copy of the default shape of the requested type. Since you can change the default shapes, you can also change the style objects that they reference.

Initially, all of the default shape objects reference the same style object. Whenever you create a new shape, it, too, references this style object. There are two ways in which you can change the style object associated with a new shape:

- n You can call a function such as `GXSetShapePen`, which makes a copy of the style object specifically for your new shape before changing its pen width.
- n You can obtain a reference to your new shape’s style object by calling the `GXGetShapeStyle` function, and then you can call a function such as `GXSetStylePen`, which does not make a copy of the style object. Instead, it affects the style object directly, which, in effect, changes the default style for all the default shapes.

By calling functions such as `GXSetShapePen` on each of the default shapes, you can create a different style object for each default shape. See the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects* for more information about default shapes.

## Curve Error

---

Curve error is the only geometric style property that doesn't affect the drawing of a shape; instead, it affects the geometric points of the shape's geometry when performing geometric operations, shape type conversions, and shape simplifications. The **curve error** property determines how far away two points must be for QuickDraw GX to consider them as separate points in these cases:

- n **Geometric operations.** QuickDraw GX guarantees that the results of the geometric operations described in the chapter “Geometric Operations” in this book, such as `GXIntersectShape` or `GXUnionShape`, have no two points closer than the value of the curve error of the target shape.
- n **Insetting shapes.** A special case of geometric operation, the `GXInsetShape` function, which is described in the chapter “Geometric Operations” in this book, can produce results with an unusually large number of geometric points. Because the inset of a quadratic Bézier curve is not a quadratic Bézier curve itself, multiple insets of tight curve shapes can cause the number of geometric points to grow dramatically. As with the other geometric operations, the result of the `GXInsetShape` function has no two consecutive points closer than the value of the curve error of the target shape.
- n **Path to polygon conversions.** The curve error also determines the maximum error when converting a path shape to a polygon (for example, with the code `GXSetShapeType(aPathShape, gxPolygonType)`). The distance between the original path and the resulting polygon is always less than the value of the curve error. If the curve error is 0, QuickDraw GX performs the path to polygon conversion simply by removing all off-curve control points, which gives a fairly rough approximation.
- n **Shape simplifications.** The functions `GXReduceShape` and `GXSimplifyShape`, which are described in more detail the chapter “Geometric Operations” in this book, perform a number of simplifications on shapes (for example, removing geometric points unnecessary to the geometry and unwinding crossed contours). In addition to their other simplifications, these functions remove all consecutive (on-curve) geometric points within a distance of less than the curve error.

The sections “Using Curve Error When Converting Paths to Polygons” on page 3-45 and “Using Curve Error When Reducing Shapes” on page 3-49 give examples of using curve error, and the section “Getting and Setting Curve Error” on page 3-114 describes the functions you can use to manipulate this style property.



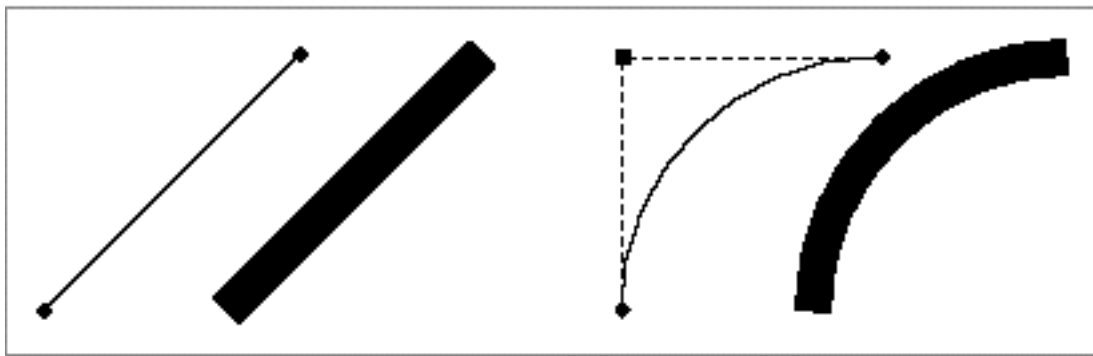
## The Geometric Pen

The contours of framed geometric shapes are drawn with the QuickDraw GX **geometric pen**. You can specify the width of this pen using the pen width property of the style object, and you can specify where to place the pen relative to the contours of a shape using the style attributes, which are described in “Style Attributes” beginning on page 3-17.

Conceptually, the QuickDraw GX geometric pen is a line that QuickDraw GX drags along the contours of the shape being drawn—always keeping it perpendicular to the contours. In effect, the geometric pen turns a framed geometry into a filled one. For example, a line shape, which is always framed, becomes the equivalent of a filled polygon after QuickDraw GX applies the geometric pen.

Figure 3-5 shows the effect of the geometric pen. This figure shows two geometries—a line geometry and a curve geometry—and how QuickDraw GX draws them with a pen width of 15.

**Figure 3-5** The QuickDraw GX geometric pen



Notice that the ends of the thick line contour and the thick curve contour in Figure 3-5 are perpendicular to the direction of the contours themselves.

Figure 3-6 shows the effect of different pen widths on a semicircular path shape.

**Figure 3-6** Differing pen widths



Setting a value of 0 for the pen width property has special meaning. Instead of indicating an infinitely thin pen, it indicates that a shape's contours should be drawn using **hairlines**—the thinnest line renderable on the device to which the shape is drawn. A hairline is always one pixel wide and is always centered about the shape's geometry.

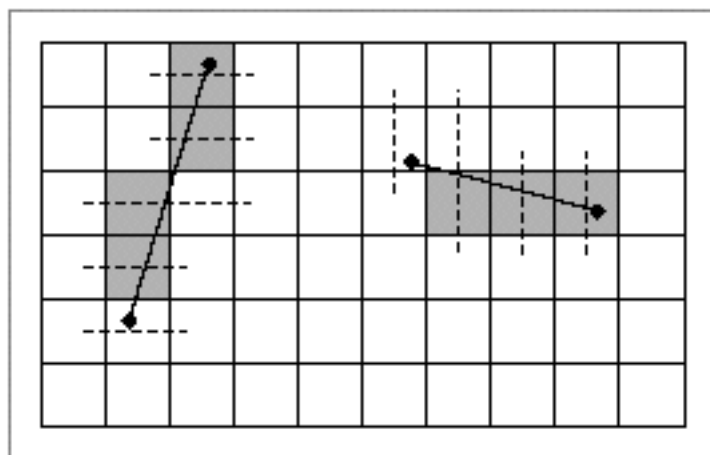
One important use of hairlines is to make point shapes visible. QuickDraw GX draws point shapes under only two conditions: if the pen width is 0, indicating a hairline point, in which case exactly one pixel is drawn, or if the point has a start cap, which is described in “Caps” beginning on page 3-23.

When drawing a hairline, QuickDraw GX uses this algorithm to determine which pixels to include:

- n If the contour being drawn is more vertical than horizontal, QuickDraw GX includes a pixel if the contour crosses the horizontal center line of the pixel.
- n If the contour being drawn is more horizontal than vertical, QuickDraw GX includes a pixel if the contour crosses the vertical center line of the pixel.

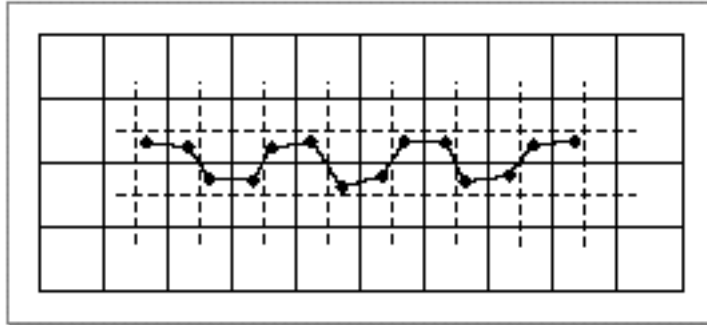
Figure 3-7 depicts this algorithm.

**Figure 3-7** Pixels included in a hairline



In extreme cases, this algorithm can cause no pixels to draw, as shown in Figure 3-8.

**Figure 3-8** A geometry with no hairline



The section “Manipulating Pen Width and Placement” on page 3-51 gives an example of using the pen width property, and the section “Getting and Setting the Pen Width” on page 3-119 describes the functions you can use to manipulate it.

## Style Attributes

The **style attributes** property of a style object contains six attributes that affect the drawing of a shape. Four of these attributes affect how QuickDraw GX places the geometric pen relative to the contours of a shape:

- n The center-frame style attribute, which is the default, indicates that the QuickDraw GX should center the geometric pen along the shape’s contours.
- n The inside-frame style attribute indicates that QuickDraw GX should position the pen along the inside of a shape’s contours.
- n The outside-frame style attribute indicates that QuickDraw GX should position the pen along the outside of shape’s contours.
- n The auto-inset style attribute affects the definition of the inside and outside of a contour.

These four attributes are discussed in the next section, “Pen Placement.”

There are also two style attributes that determine whether the geometric points of a shape are constrained to a grid when the shape is drawn:

- n The source-grid style attribute constrains the geometric points of a shape to integer values before applying the shape’s style and transform information.
- n The device-grid style attribute constrains the geometric points of a shape to integer pixel positions after applying the shape’s style and transform information.

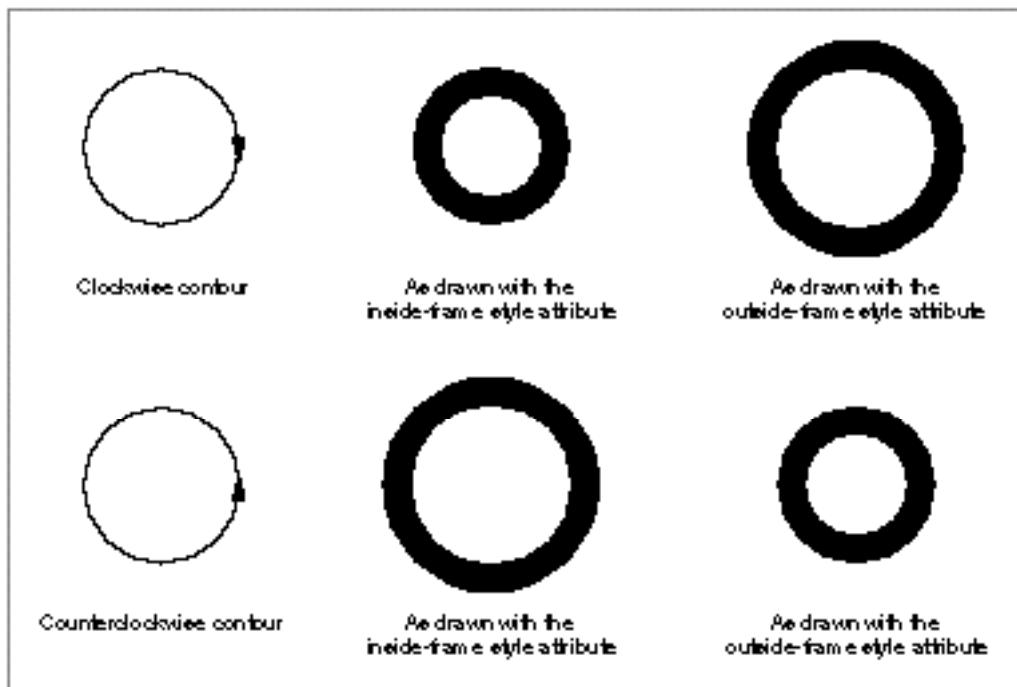
These two attributes are discussed in the section “Grids” beginning on page 3-20.

## Pen Placement

You can use the center-frame, inside-frame, and outside-frame style attributes to specify where QuickDraw GX should position the pen with respect to the shape's geometry. QuickDraw uses these attributes to position the pen, which also affects the placement of dashes and how dashes are clipped. For some examples, see "Insetting Dashes" beginning on page 3-73 and "Combining Caps, Joins, Dashes, and Patterns" beginning on page 3-91.

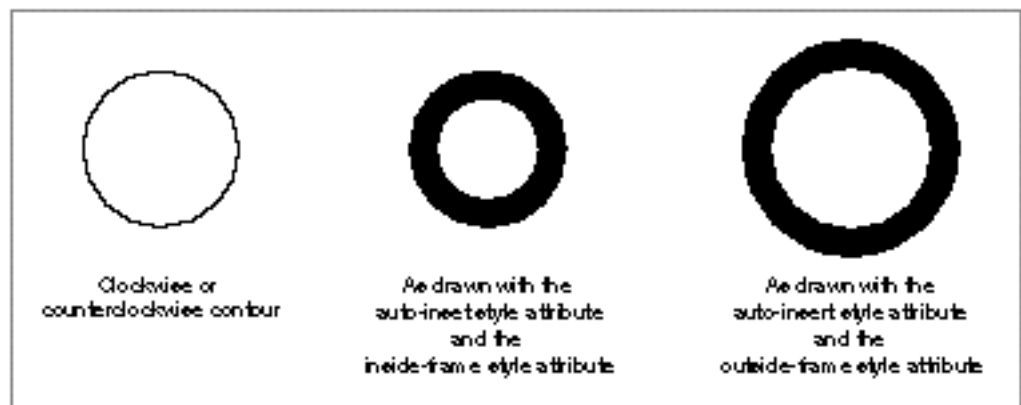
Figure 3-9 shows the results of these style attributes. Notice that QuickDraw GX considers contour direction when determining which side of a contour is the inside: the right side of the contour is the inside, while the left side of the contour is the outside.

**Figure 3-9** Pen placement



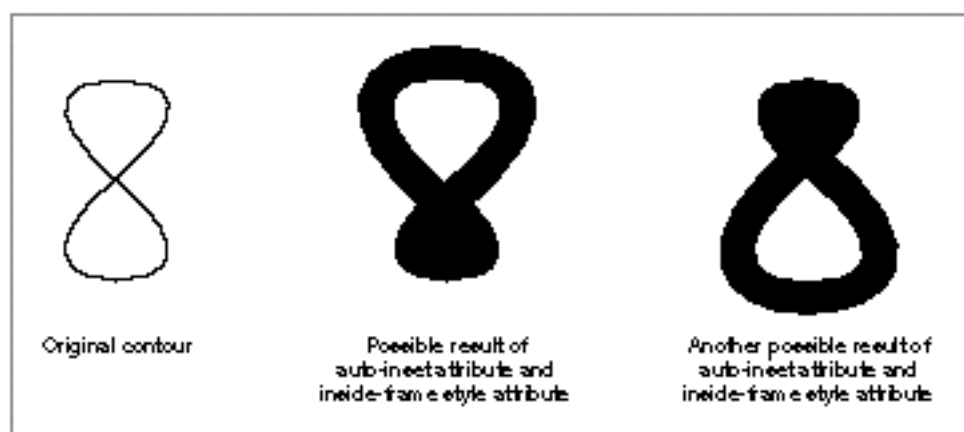
QuickDraw GX also provides the auto-inset style attribute, which allows you to specify that QuickDraw GX should ignore contour direction when determining which side of a contour is the inside. When you set this style attribute, QuickDraw GX determines the true inside of a contour, rather than using the right side as the inside. Figure 3-10 shows the effect of setting the auto-inset style attribute for the shapes depicted in Figure 3-9.

**Figure 3-10** Effect of the auto-inset style attribute



When a contour crosses over itself, the results of setting the auto-inset style attribute are unpredictable, as the contour has no true inside (or, actually, has multiple true insides). For the figure-eight shape in Figure 3-11, setting the `gxAutoInsetStyle` and the `gxInsideFrameStyle` style attributes could lead to one of two results.

**Figure 3-11** Effect of the auto-inset and inside-frame style attributes for a crossed contour



To ensure that setting the auto-inset style attribute behaves as you would expect, you need to call the `GXSimplifyShape` function, which is described in the chapter “Geometric Operations” in this book. This function redefines the shape’s geometry to eliminate crossed contours, as shown in Figure 3-12.

**Figure 3-12** Eliminating crossed contours



The section “Manipulating Pen Width and Placement” on page 3-51 gives an example of specifying pen placement. The section “Style Attributes” on page 3-98 defines the style attributes enumeration, and the section “Getting and Setting Style Attributes” on page 3-109 describes the functions you can use to manipulate them.

## Grids

From the initial geometry specification to the final image rendering, each QuickDraw GX shape exists in a number of different coordinate spaces. You describe a shape’s geometry in geometry space, the style and transform modifications happen in local space, the shape then exists in one or more view ports’ global spaces, and the shape is finally rendered in the pixels of a view device’s device space.

In each of these coordinate spaces, QuickDraw GX allows fractional coordinate values. When you specify points in a shape’s geometry, you are not limited to integer values, such as (1, 1) or (–10, 10). Instead, you can specify that the shape’s geometric points fall between integral positions in the geometry space’s coordinate grid, for example (0.5, 0.5). During each transformation of the shape from geometry to rendering, QuickDraw GX maintains fractional coordinate values.

The style attributes property of a style object contains two flags that allow you to suppress fractional coordinate values—that is, these flags allow you to constrain a shape’s geometric points to integer coordinate values in the different coordinate systems.

The source-grid style attribute indicates that QuickDraw GX should constrain the shape's geometric points to integral positions on the local space grid, before making the style and transform modifications.

The device-grid style attribute indicates that QuickDraw GX should constrain the shape's geometric points to integral positions (that is, pixel positions) on the device space grid, after making style, transform, and view port modifications.

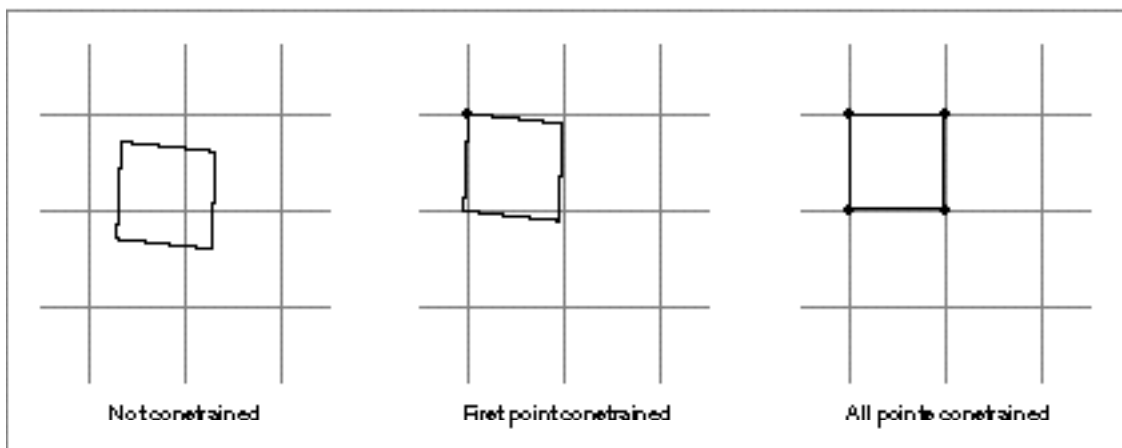
#### Note

These style attributes only affect a shape while it is being drawn. They do not affect the geometric points you specify in the original shape geometry. u

To constrain a shape to integral positions on a coordinate space's grid, QuickDraw GX moves the entire shape (that is, all the shape's geometric points) so that the shape's first geometric point lies on the nearest grid position, and then moves each remaining geometric point to the nearest grid position.

Figure 3-13 depicts the grid-constraining algorithm.

**Figure 3-13** Constraining shapes to grids



The sections “Constraining Shape Geometries to Grids” on page 3-40 and “Constraining Shapes to Device Grids” on page 3-42 give examples of the grid-constraining style attributes. The section “Style Attributes” on page 3-98 defines the style attributes enumeration and the section “Getting and Setting Style Attributes” beginning on page 3-109 describes the functions you can use to manipulate them.

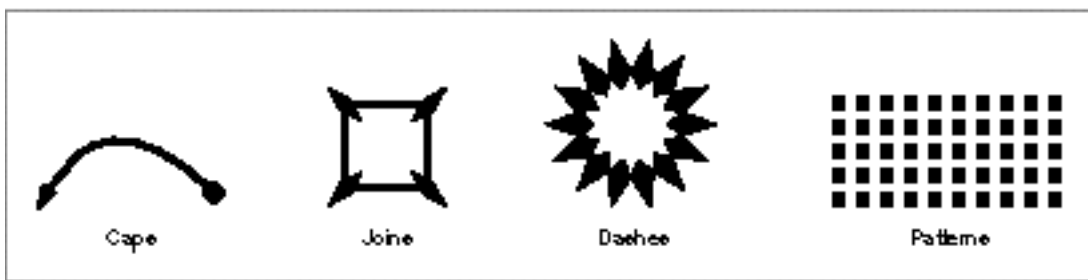
## Interactions Between Caps, Joins, Dashes, and Patterns

The cap, join, dash, and pattern properties of the style object allow you to change the way QuickDraw GX draws the contours of a shape. The cap and join properties allow you to place arbitrary shapes on the geometric points of a shape's contours. For example, you can place arrow heads at the ends of a line, or you can put rounded edges at the corners of a rectangle. The dash property allows you to dash the contours of one shape with another shape. For example, you could dash a line with a circular path shape to get a dotted line.

The pattern property allows you to fill a shape (or the frame of a shape drawn with a thick pen width) with a repeated pattern of another shape. For example, you could fill a large square shape with a pattern of small squares to get a checkerboard.

Figure 3-14 shows some of the stylistic variations possible with caps, joins, dashes, and patterns.

**Figure 3-14** Caps, joins, dashes, and patterns



There is one important rule that applies to all four of these properties: Cap shapes, join shapes, dash shapes, and pattern shapes must all be in their primitive form. When QuickDraw GX uses a cap, join, dash, or pattern shape, it ignores the stylistic variations of that shape. If you want a cap, join, dash, or pattern shape to have stylistic variations itself, you must first incorporate those stylistic variations into the shape using the `GXPrimitiveShape` function.

As an example, specifying a line shape with a thick pen (like the one in Figure 3-3 on page 3-9) as a cap, join, dash, or pattern shape may produce an unexpected result or post an error, since the shape is not in its primitive form. However, if you use the `GXPrimitiveShape` function, you can convert the line to a filled polygon, which is a perfectly acceptable cap, join, dash, or pattern shape.

The sections “The Cap Structure” on page 3-99, “The Join Structure” on page 3-101, “The Dash Structure” on page 3-103, and “The Pattern Structure” on page 3-106 discuss the types of shapes appropriate to use as caps, joins, dashes, and patterns.



As another example, a polygon with zero contours is not an acceptable cap, join, dash, or pattern shape, as it is not in its primitive form. Similarly, any shape with the no-fill shape fill is not in its primitive form. However, the empty shape, which is in its primitive form, is an acceptable cap, join, dash, or pattern shape. You can find more information about polygon shapes, polygon contours, and empty shapes in Chapter 2, “Geometric Shapes,” in this book.

You can always be sure your cap, join, dash, or pattern shape is in the correct form by calling the `GXPrimitiveShape` function, which is described in Chapter 4, “Geometric Operations,” before setting the corresponding style property.

As for typographic shapes, text and layout shapes are not in their primitive form, but glyph shapes are acceptable as cap, join, dash, and pattern shapes so long as they have no text face or tags and do not have caps, joins, dashes, or patterns themselves.

For more information, see *Inside Macintosh: QuickDraw GX Typography*.

You can use bitmap shapes as patterns, but not as caps, joins, or dashes, and you cannot use picture shapes for caps, joins, dashes, or patterns.

The next few sections describe caps, joins, dashes, and patterns in more detail.

## Caps

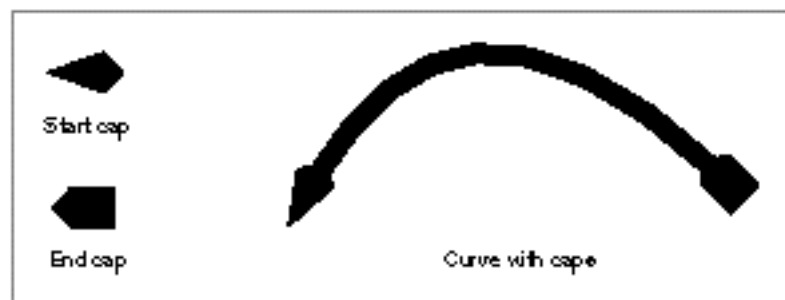
QuickDraw GX allows you to specify **cap shapes**—what to draw at the start and at the end of a shape’s contours. In particular, you can specify a start cap for any point shape, and you can specify a start cap and an end cap for any line, curve, polygon, or path shape that has open-frame shape fill.

In fact, the only way to draw a point shape is to specify a start cap for it (unless its style object has a pen width property with a value of 0, in which case QuickDraw GX draws the point shape as a single pixel).

QuickDraw GX uses the **cap property** of a shape’s style object to store information about the start cap and end cap of the shape.

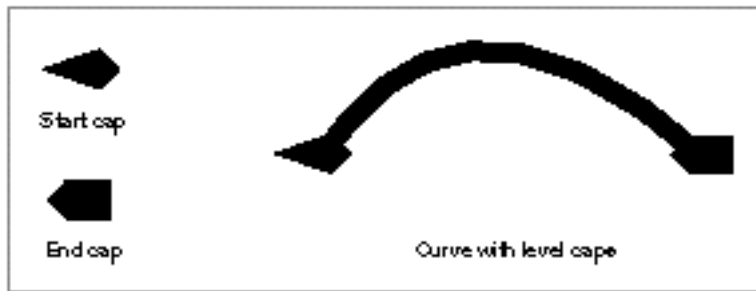
Figure 3-15 shows how QuickDraw GX adds a cap to a contour by centering the cap shape at the end of the contour, scaling the cap shape by the pen width, and rotating the cap shape to match the slope of the contour.

**Figure 3-15** A shape with caps



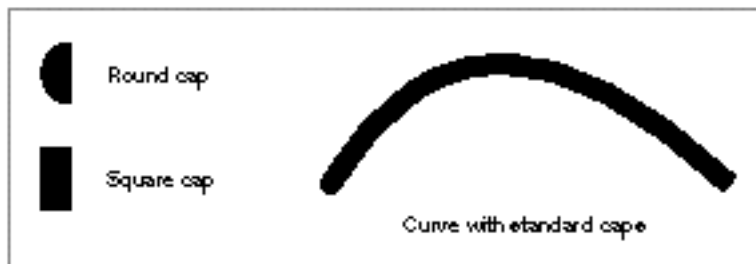
The cap property of a style object includes a **cap attributes** field, which allows you to specify **level caps**—caps that QuickDraw GX does not rotate to match the slope of the contour—as shown in Figure 3-16.

**Figure 3-16** A shape with level caps



You can create two standard cap types by specifying half a square or a semicircle for the start cap or end cap shape, as shown in Figure 3-17.

**Figure 3-17** Standard cap shapes



The sections “Adding Caps to a Shape” on page 3-57 and “Adding Standard Caps to a Shape” on page 3-59 give examples of using the cap property, the sections “The Cap Structure” on page 3-99 and “Cap Attributes” on page 3-101 describe the data structures used to store information about caps, and the section “Getting and Setting Caps” beginning on page 3-123 describes the functions you can use to manipulate caps.

## Joins

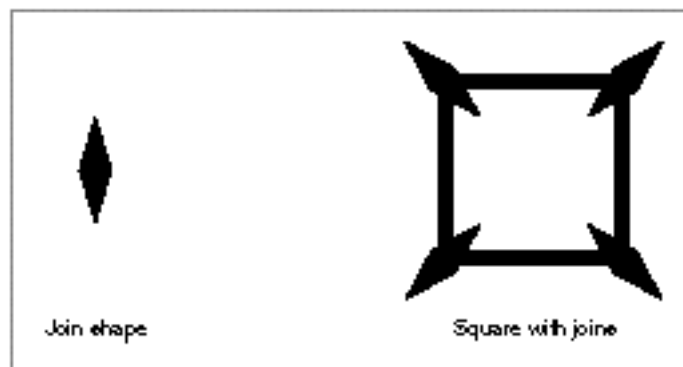
QuickDraw GX allows you to specify a **join shape** to be drawn at the corners of another shape's contours. In particular, you can specify a join shape for any rectangle, polygon, or path shape that has an open-frame shape fill or a closed-frame shape fill:

- n For shapes with the closed-frame shape fill, QuickDraw GX draws the specified join shape at every on-curve geometric point of each contour.
- n For shapes with the open-frame shape fill, QuickDraw GX draws the specified join shape at every on-curve geometric point between the first point and the last point of each contour.

QuickDraw GX uses the **join property** of a shape's style object to store information about the joins of a shape.

Figure 3-18 shows how QuickDraw GX adds a join to a contour by centering the join shape on the on-curve geometric points, scaling the join shape by the pen width, and rotating the join shape to match the mid-angle of the two line segments that make up the corner.

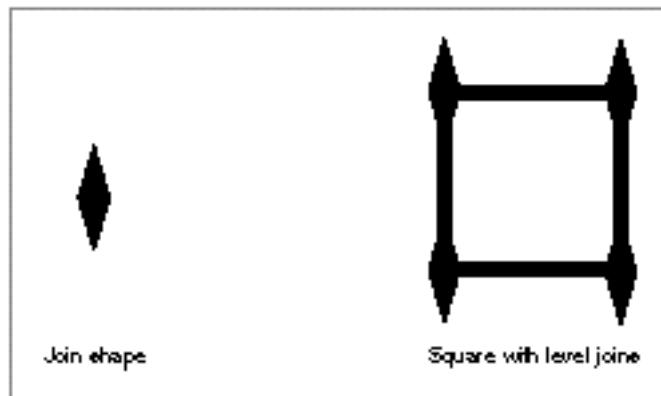
**Figure 3-18** A shape with joins



## Geometric Styles

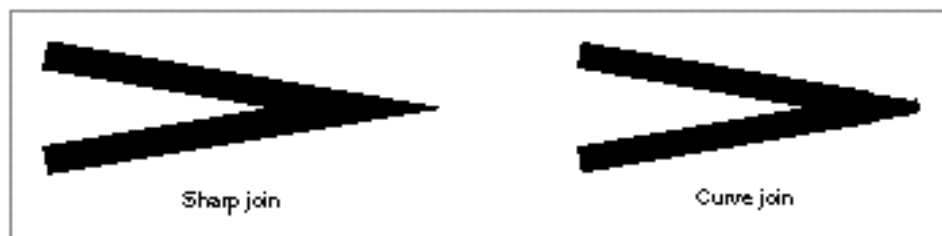
The join property of a style object includes a **join attributes** field, which allows you to specify **level joins**—joins that QuickDraw GX does not rotate to match the slope of the contour—as shown in Figure 3-19.

**Figure 3-19** A shape with level joins



You can also use the join attributes to specify two types of standard joins—sharp joins and curve joins, as shown in Figure 3-20.

**Figure 3-20** Standard joins



For sharp joins, QuickDraw GX allows you to specify a **miter**—the maximum distance between the actual corner of a shape’s geometry and the corner as drawn, as shown in Figure 3-21.

**Figure 3-21** Sharp join with miter



The sections “Adding Joins to a Shape” on page 3-61 and “Adding Standard Joins to a Shape” on page 3-64 give examples of using the join property, the section “The Join Structure” on page 3-101 and “Join Attributes” on page 3-102 describe the data structures used to store information about joins, and the section “Getting and Setting Joins” on page 3-129 describes the functions you can use to manipulate them.

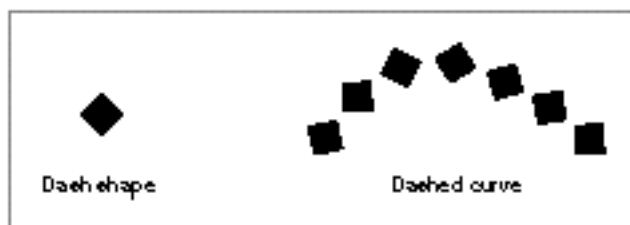
## Dashes

With QuickDraw GX, you can specify that framed shapes should be drawn with dashed, instead of solid, contours. In particular, you may specify a **dash shape** for any line, curve, rectangle, polygon, or path shape that has an open-frame shape fill or a closed-frame shape fill.

QuickDraw GX uses the **dash property** of a shape’s style object to store information about how to dash the shape.

Figure 3-22 shows how QuickDraw GX dashes a contour by placing copies of the dash shape along the contour at regular intervals, and rotating the dash shape to match the slope of the contour.

**Figure 3-22** A dashed shape



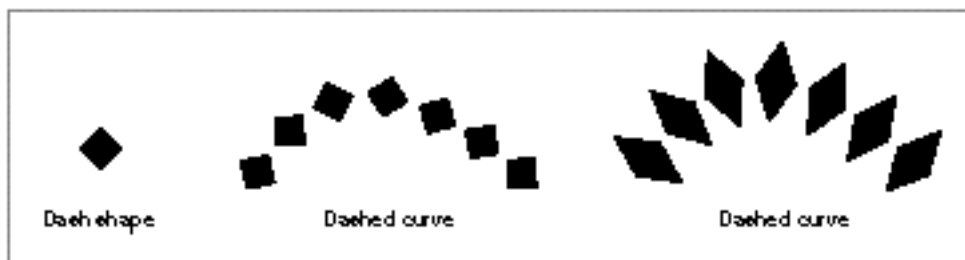
When drawing a dashed shape, QuickDraw GX automatically scales the dash shape up by the pen width of the dashed shape. However, unlike cap and joins, QuickDraw GX scales dashes only *perpendicularly to the dashed contour*.

For example, if the height of the dash shape is 1.0, then QuickDraw GX draws the dashes with a height equal to the dashed shape's pen width. If the height of the dash shape is 2.0, then QuickDraw GX draws the dashes with a height equal to twice the dashed shape's pen width.

Since the dash shape is scaled up by the pen width of the dashed shape, QuickDraw GX provides a way for you to scale the dash down, as well, by providing a scaling factor, called the **dash scale**, in one of the fields of the dash structure. QuickDraw GX multiplies the height of the dash by the pen width and then divides by the dash scale.

Figure 3-23 shows the effect of different pen widths on the same dash shape. In this example, the dash shape has height of 10.0 (its y-coordinates span from -5.0 to 5.0). The shape being dashed is a curve. The curve is shown first with a pen width of 10.0 and a dash scale of 10.0, which keeps the dimensions of the dash shape unchanged. The curve is then shown with a pen width of 20.0 and a dash scale of 10.0, which doubles the size of the dash shape in the y-coordinate direction.

**Figure 3-23**     Scaling a dash shape



#### Note

Glyph shapes are an exception to this scaling rule. If the dash shape is a glyph shape, QuickDraw GX does not scale the dashes (which in this case would be glyphs) to the dashed shape's pen width. u

Notice that the position of a dash shape in the coordinates of its geometry space is significant. For example, if the y-coordinates of the geometry of a dash shape span from 1.0 to 2.0, then QuickDraw GX draws the dashes at a distance of one pen width to the outside of the dashed contour (if the dash scale is 1.0). If the lowest y-coordinate of a dash shape is 2.0, then QuickDraw GX draws the dashes at a distance of two pen widths to the outside of the contour (if the dash scale is 1.0).

If the y-coordinates of the geometry of a dash shape are large enough and the scaling factor you provided in the dash structure is small enough, the dashes may exceed the pen width of the dashed shape. QuickDraw GX provides the clip dash attribute to indicate that QuickDraw GX should clip the dashes to the pen width, as illustrated in Figure 3-24.

**Figure 3-24** Effect of the clip dash attribute

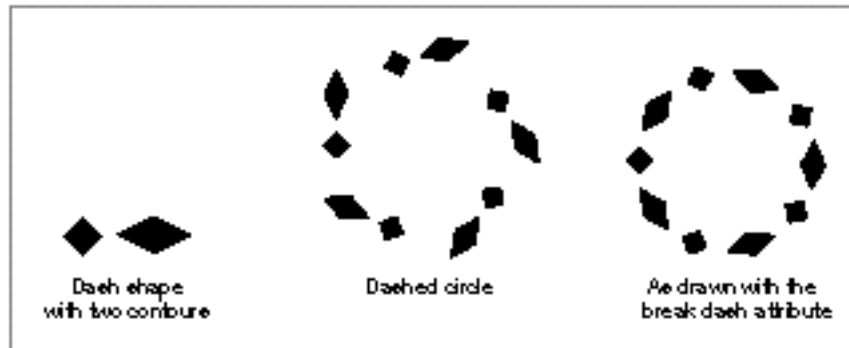


Setting the clip dash attribute causes some intricate interactions among dashes, caps, joins, and patterns. See “Interactions Between Caps, Joins, Dashes, and Patterns” on page 3-33 for more information.

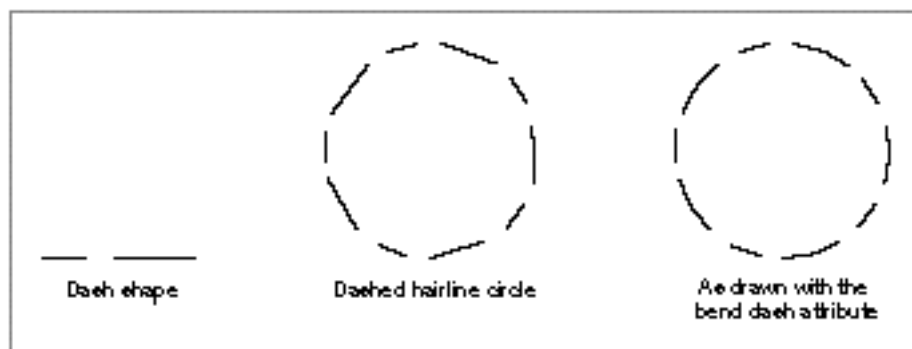
QuickDraw GX also allows you to control how far apart the dashes appear from one another, which is called the **dash advance**, and how far into the dash shape the dashing should start, which is called the **dash phase**.

The dash advance is the distance between the start of one dash shape and the start of the next dash shape along the contour. The dash phase indicates where the first dash should fall on a contour; it is a percentage of the dash advance.

When a dash shape has multiple contours, it is possible for the dashes not to fall on the contours of the dashed shape. For this situation, QuickDraw GX provides the break dash attribute, which indicates that each contour of the dash should be rotated and placed separately on the dashed shape’s contours. Figure 3-25 depicts the result of setting this dash attribute.

**Figure 3-25** Effects of breaking a dash

Finally, QuickDraw GX provides a dashing feature that works only when dashing hairline contours. In this case, you can set the bend dash attribute, which indicates that QuickDraw GX should wrap the dash to fit the dashed contour exactly, as shown in Figure 3-26.

**Figure 3-26** Effects of bending a dash



The following sections give examples of dashing:

- n “Dashing a Shape” on page 3-66
- n “Adjusting Dashes to Fit Contours” on page 3-70
- n “Insetting Dashes” on page 3-73
- n “Breaking and Bending Dashes” on page 3-74
- n “Wrapping Text to a Contour” on page 3-80
- n “Determining Dash Positions” on page 3-81

The section “The Dash Structure” on page 3-103 and “Dash Attributes” on page 3-105 describe the data structures used to store and communicate information about dashes, and the section “Getting and Setting Dashes” on page 3-134 describes the functions you can use to manipulate the dash property.

## Patterns

---

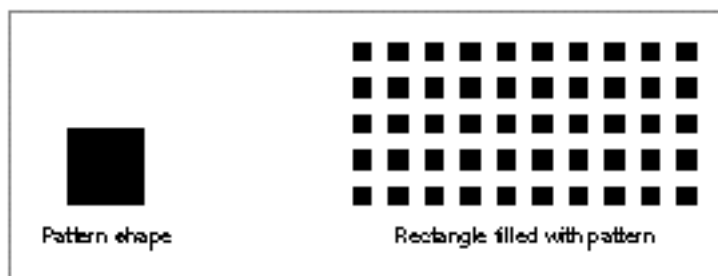
With QuickDraw GX, you can specify that certain shapes be filled with a pattern. For shapes with solid shape fills, QuickDraw GX fills the shape by repeating a pattern shape that you specify.

You can also pattern framed shapes. For example, if you pattern a rectangle shape with a closed-frame shape fill and a pen width of 20.0, QuickDraw GX would fill the frame of the rectangle with the pattern. See the section “Interactions Between Caps, Joins, Dashes, and Patterns” on page 3-33 for more intricate examples.

QuickDraw GX uses the **pattern property** of a shape’s style object to store information about how to pattern the shape.

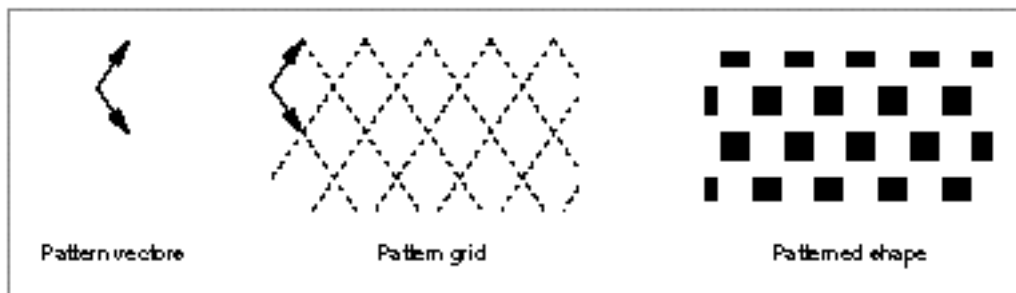
Figure 3-27 shows how QuickDraw GX patterns a shape by filling the shape with copies of another shape, called the **pattern shape**, placed according to a regular grid that you specify.

**Figure 3-27** A shape with a pattern



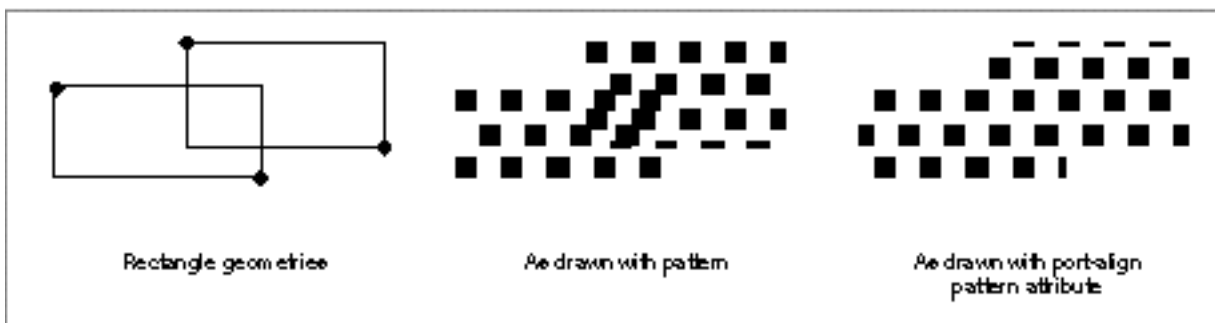
In addition to specifying the shape to use as the pattern shape, you also specify the **pattern grid** on which to place the pattern, as shown in Figure 3-28.

**Figure 3-28** Pattern placed on a nonrectilinear grid



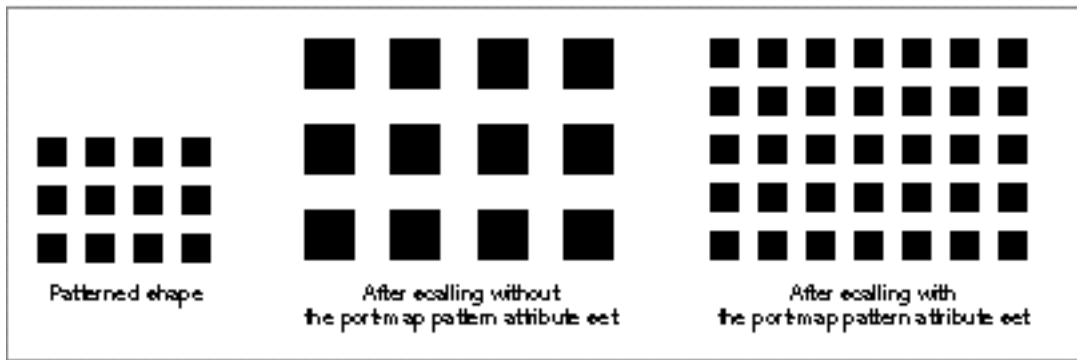
In addition, QuickDraw GX provides you with two **pattern attributes**: the port-align pattern attribute and the port-map pattern attribute. Setting the port-align pattern attribute allows you to specify that QuickDraw GX should align the pattern with the view device instead of with the geometry of the patterned shape. Figure 3-29 shows the effect of setting this attribute.

**Figure 3-29** Effects of the port-align pattern attribute



The port-map pattern attribute indicates that the pattern shape should not be affected by transformations to the patterned shape. For example, if you set this pattern attribute, scaling the patterned shape by a factor of two does not also scale the pattern shape by a factor of two; instead, more of the pattern is shown. Figure 3-30 shows the effect of setting this attribute.

**Figure 3-30** Effects of the port-map pattern attribute



The sections “Adding a Pattern to a Shape” on page 3-86 and “Determining Pattern Positions” on page 3-88 give examples of using patterns. The section “The Pattern Structure” on page 3-106 and “Pattern Attributes” on page 3-107 describes the data structures used to store information about patterns, and the section “Getting and Setting Patterns” on page 3-142 describes the functions you can use to manipulate patterns.

### Interactions Between Caps, Joins, Dashes, and Patterns

The previous four sections show the results of adding a cap, a join, a dash, or a pattern to a QuickDraw GX shape. This section discusses how these stylistic variations interact when you add more than one of them at a time to the same shape.

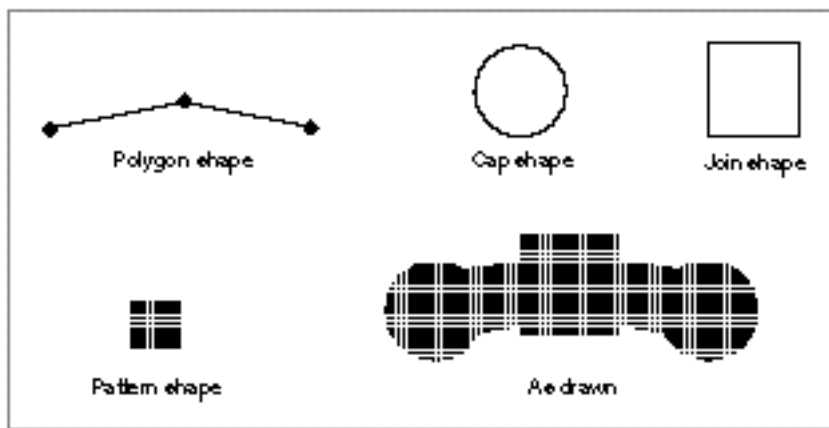
In general, these elements interact differently in each of these three cases:

- n the shape does not have a dash but has one or more of the other three stylistic variations
- n the shape does have a dash but the clip dash attribute is not set
- n the shape does have a dash and the clip dash attribute is set

## Geometric Styles

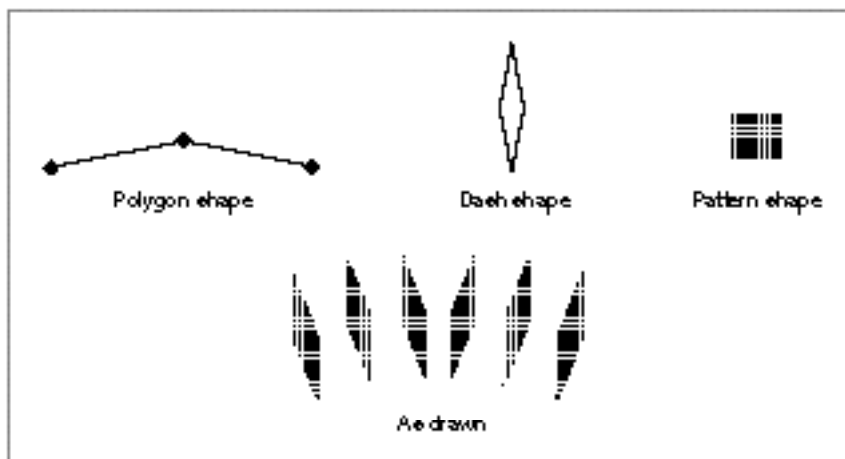
When a shape has a cap and a join, QuickDraw GX adds the caps to the beginnings and ends of the shape's contours, and adds the joins to the other on-curve geometric points of the shape's contours. If the shape also has a pattern, QuickDraw GX draws this pattern throughout the shape's frame as well as the shape's caps and joins, as shown in Figure 3-31.

**Figure 3-31** A shape with a cap, join, and pattern



If a shape has a dash, but the clip dash attribute is not set, QuickDraw GX *ignores the caps and joins of the shape*. However, if the shape has a pattern, QuickDraw GX does draw the pattern throughout the dashes, as shown in Figure 3-32.

**Figure 3-32** A shape with a dash and a pattern



Finally, if the shape has a dash and the clip dash attribute is set, QuickDraw GX does not ignore the caps and joins. Instead, the cap shapes and the join shapes are added to the clip shape that QuickDraw GX uses to clip the dashes. Patterns are not allowed in this case. Figure 3-33 shows the interaction of a cap, join, and clipped dash.

**Figure 3-33** A shape with a clipped dash and a cap and join



The section “Combining Caps, Joins, Dashes, and Patterns” beginning on page 3-91 give examples of the interactions between caps, joins, dashes, and patterns.

## Using Geometric Styles

This section shows you how to use styles to add stylistic variations to geometric shapes. In particular, this section show you how to

- n create a style object, alter its properties, and associate the style with a shape
- n alter the properties of a style object already associated with a shape
- n constrain shapes to grids
- n use curve error when approximating paths with polygons and when reducing shapes
- n manipulate pen width and placement

## Geometric Styles

- n add caps to a shape, including round and square caps
- n add joins to a shape, including standard round and sharp joins
- n dash a shape
- n adjust dashes to fit contours
- n bend and break dashes
- n wrap text by using glyphs as a dash shape
- n determine dash positions
- n add a pattern to a shape and determine pattern positions
- n combine caps, joins, dashes, and patterns

## Associating Styles With Shapes

---

QuickDraw GX provides two basic methods of altering stylistic information for shapes:

- n using functions that operate on style objects directly
- n using functions that operate on style objects indirectly through shape objects

The first category of functions require you to provide a reference to a style object, which you can obtain by using the `GXNewStyle` function to create a new style object, or by using the `GXGetShapeStyle` function to obtain a reference to an existing style object. (The `GXNewStyle` and `GXGetShapeStyle` functions are described in *Inside Macintosh: QuickDraw GX Objects*.)

Once you have a reference to a style object, you can use this category of functions to manipulate the style's properties; for example, you can use the `GXSetStylePen` function to change the pen width of the style.

If you obtained the reference to the style object using the `GXGetShapeStyle` function, then the style is already associated with a shape—in fact, it may be shared amongst many shapes. Modifications you make to the style's properties will apply to all shapes that share the style.

However, if you created the style object using the `GXNewStyle` function, you must then associate the style with a shape for the style modifications to have any effect. You can associate a style with a shape using the `GXSetShapeStyle` function, as shown in Listing 3-1. The `GXSetShapeStyle` function is described in *Inside Macintosh: QuickDraw GX Objects*.

---

**Listing 3-1** Adding style information by directly manipulating a style object

```
void MakeThickPenStyle(void)
{
    gxShape aRectangleShape;
    gxStyle aThickPenStyle;

    static gxRectangle rectangleGeometry = {ff(50), ff(50),
                                           ff(200), ff(200)};

    aRectangleShape = GXNewRectangle(&rectangleGeometry);
    GXSetShapeFill(aRectangleShape, gxClosedFrameFill);

    aThickPenStyle = GXNewStyle();
    GXSetStylePen(aThickPenStyle, ff(30));

    GXSetShapeStyle(aRectangleShape, aThickPenStyle);
    GXDisposeStyle(aThickPenStyle);

    GXDrawShape(aRectangleShape);

    GXDisposeShape(aRectangleShape);
}
```

The `MakeThickPenStyle` sample function creates a rectangle shape and sets its shape fill to the closed-frame shape fill, making it a framed rectangle. The sample function then creates a new style object using the `GXNewStyle` function, which creates a style object with properties set to the standard initialized values. The owner count of this style object is 1, corresponding to the reference contained in the `aThickPenStyle` variable. The sample function then alters the pen width of the new style using the `GXSetStylePen` function.

To associate the style with the rectangle shape, the sample function calls the `GXSetShapeStyle` function. This function disposes of the style previously referenced by the rectangle shape, stores a reference to the new style in the rectangle shape object, and increments the style's owner count—there are now two references to the style: one in the sample function's local variable, and one in the rectangle shape.

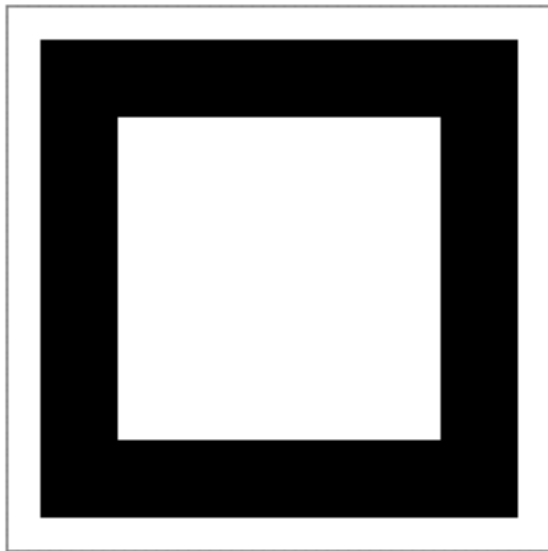
Finally, the sample function disposes of the style, which indicates that the reference to the style stored in the local variable `aThickPenStyle` is no longer needed.

`QuickDraw GX` decrements the owner count of the style, which becomes 1, corresponding to the reference contained in the rectangle shape.

Finally, the sample function draws the rectangle, which appears as in Figure 3-34.

---

**Figure 3-34** Rectangle with thick pen



The second method of altering styles involves functions that operate on style objects indirectly through the shape objects that reference them.

When using this category of function, you need only provide a reference to the shape whose style information you want to change. `QuickDraw GX` finds the associated style object and alters the appropriate style property for you.



In fact, QuickDraw GX provides one further level of service with this category of functions. If the shape that you specify is sharing its style with other shapes, QuickDraw GX first makes a copy of the style object, associates the copy with the shape you specified, and then alters the appropriate property of the copy.

Listing 3-2 shows an alternate approach to creating the thick-framed rectangle from Listing 3-1.

---

**Listing 3-2**      Manipulating style information indirectly

```
void MakeThickRectangle(void)
{
    gxShape aRectangleShape;

    gxRectangle rectangleGeometry = {ff(50), ff(50),
                                     ff(200), ff(200)};

    aRectangleShape = GXNewRectangle(&rectangleGeometry);
    GXSetShapeFill(aRectangleShape, gxHollowFill);

    GXSetShapePen(aRectangleShape, ff(30));

    GXDrawShape(aRectangleShape);

    GXDisposeShape(aRectangleShape);
}
```

As in Listing 3-1, this sample function creates a new framed rectangle shape. However, instead of creating a style object, altering the pen width property of the style object with the `GXSetStylePen` function, and associating the style with the rectangle shape, this sample function uses the `GXSetShapePen` function to accomplish those tasks in one step.

Since the rectangle shape is a new shape, it shares its style object with other shapes—the default rectangle shape at the very least. The `GXSetShapePen` function notices that the rectangle shape's style is shared, so it makes a copy of this style, associates the copy with the rectangle shape, and alters the pen width property of this copy.

The result of this sample function looks exactly the same as the result of the previous sample function, shown in Figure 3-34.

For simplicity, the rest of the sample functions in this chapter use the second method for altering style properties.

## Constraining Shape Geometries to Grids

---

The source-grid style attribute (`gxSourceGridStyle`) allows you to specify that QuickDraw GX should constrain the coordinates of a shape's geometry to integer positions before applying the shape's style and transform. Setting this style attribute does not actually change the information stored in the shape's geometry—instead, QuickDraw GX reinterprets the shape's geometry when drawing the shape.

If a shape has no style or transform modifications, setting this style attribute has the effect of snapping the shape to a 1/72-inch grid—an effect that is visible only on high-resolution devices. However, if the shape has style or transform modifications, setting this style attribute might have visible effects even on lower-resolution devices.

For example, you can use the source-grid style attribute in combination with a scaling transform to achieve the effect of constraining a shape to a grid much larger than 1/72 inch. The sample function in Listing 3-3 shows how to use this style attribute to constrain a shape to a half-inch grid.

---

**Listing 3-3**      Constraining a shape to a half-inch grid

```
void ConstrainShapeToGrid(void)
{
    gxMapping scaleToHalfInches;
    static long veeGeometry[] = {1, /* number of contours */
                                3, /* number of points */
                                fl(1.2), fl(1.1),
                                fl(2.9), fl(2.8),
                                fl(5.2), fl(.9)};

    gxShape aVeeShape;

    aVeeShape = GXNewPolygons((gxPolygons *) veeGeometry);
    GXSetShapeFill(aVeeShape, gxOpenFrameFill);
    GXSetShapePen(aVeeShape, fl(5.0 * (1.0/36.0)));
```

## Geometric Styles

```

GXResetMapping(scaleToHalfInches);
GXScaleMapping(scaleToHalfInches, ff(36), ff(36),
               ff(0), ff(0));
GXSetShapeMapping(aVeeShape, scaleToHalfInches);

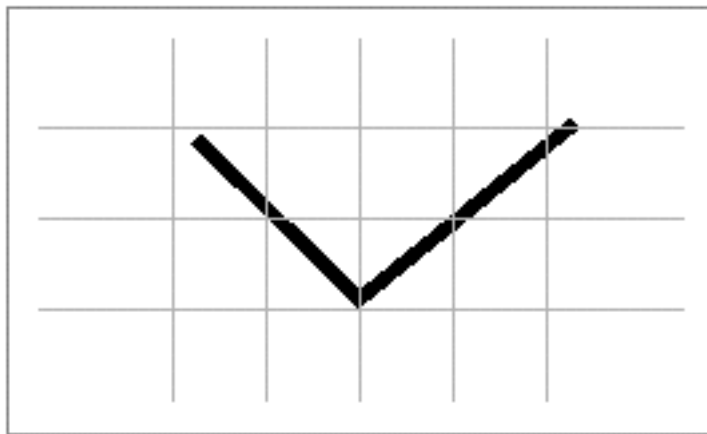
GXDrawShape(aVeeShape);

GXDisposeShape(aVeeShape);
}

```

This sample function defines a small, irregular, V-shape geometry and scales the shape up by 36 points, or half an inch. The pen width is set to 5.0 (divided by 36.0 to counteract the scaling). The result of this sample function is shown in Figure 3-35.

**Figure 3-35** Scaled, but not constrained, V shape



Notice that before QuickDraw GX applies the mapping, the coordinates of the shape's geometry represent points, whereas after QuickDraw GX applies the mapping, the coordinates of the shape's geometry effectively represent half inches.

If you set the source-grid style attribute by adding this line of code to the sample function:

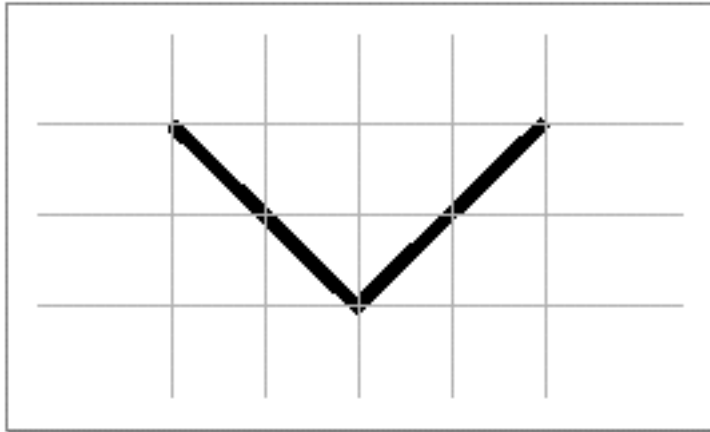
```

GXSetShapeStyleAttributes(aVeeShape, gxSourceGridStyle);

```

QuickDraw GX constrains the coordinates of the shape's geometry to the nearest integer position before applying the mapping. Therefore, after the mapping, the shape's geometric points lie on a half-inch grid, as shown in Figure 3-36.

**Figure 3-36** Constrained V shape



The sample function in this section uses some concepts from other parts of QuickDraw GX. For more information about scaling, mappings, and transforms, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

For more information about the `gxSourceGridStyle` style attribute, see “Style Attributes” on page 3-98.

## Constraining Shapes to Device Grids

QuickDraw GX provides the device-grid style attribute (`gxDeviceGridStyle`), which allows you to constrain the geometric points of a shape to integer positions *after* the style, transform, and view modifications have been made.

This style attribute constrains the geometric points of a shape to the nearest integer pixel position on the device to which the shape is rendered. Unlike the source-grid style attribute, the device-grid style attribute never drastically affects the position of the shape. However, for shapes that do not have the device-grid attribute set, QuickDraw GX makes minor modifications when drawing contours whose geometric points lie between pixels; you can use the device-grid style attribute to override these modifications, which typically produces better-looking results.

The sample function in Listing 3-4 creates a star-shaped polygon and rotates it 28 degrees, which causes its geometric points to lie between integer positions.

---

**Listing 3-4**     Creating a shape with fractional geometric point positions

```
void ConstrainShapeToDeviceGrid(void)
{
    long  starGeometry[] = {1, /* number of contours */
                           9, /* number of points */
                           ff(40), ff(40),
                           ff(50), ff(20),
                           ff(60), ff(40),
                           ff(80), ff(50),
                           ff(60), ff(60),
                           ff(50), ff(80),
                           ff(40), ff(60),
                           ff(20), ff(50),
                           ff(40), ff(40),
                           };

    gxShape aStar;

    aStar = GXNewPolygons((gxPolygons *) starGeometry);
    GXSetShapeFill(aStar, gxOpenFrameFill);

    RotateShapeAboutCenter(aStar, ff(28));
    GXDrawShape(aStar);

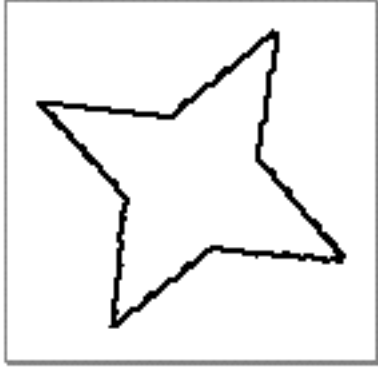
    GXDisposeShape(aStar);
}
```

## Geometric Styles

Because the geometric points of the rotated star do not lie on integer positions, QuickDraw GX does not draw the contours of the star with the most visually appealing lines; instead, it makes minor adjustments to reflect the fractional part of the geometric point coordinates as shown in Figure 3-37.

---

**Figure 3-37** Rotated star not constrained to device grid (magnified 200 percent)

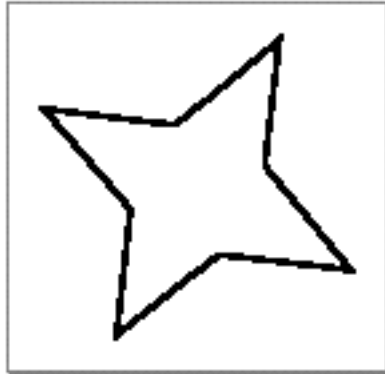


If you constrain the star shape to the device grid by adding this line of code to the sample function:

```
GXSetShapeStyleAttributes(aStar, gxDeviceGridStyle);
```

QuickDraw GX constrains the shape's geometric points to the device grid before choosing the pixels to represent the shape's contours, which creates better-looking lines, as shown in Figure 3-38.

---

**Figure 3-38** Rotated star constrained to device grid (magnified 200 percent)

The sample function in this section uses some concepts from other parts of QuickDraw GX. For more information about rotating, mappings, and transforms, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

For more information about the `gxDeviceGridStyle` style attribute, see “Style Attributes” on page 3-98.

---

## Using Curve Error When Converting Paths to Polygons

You can use the curve error property of the style object in a variety of situations—for example, when approximating a path shape (which includes curves) with a polygon shape (which includes only straight lines).

The `GXSetShapeType` function, which is described in full in *Inside Macintosh: QuickDraw GX Objects*, allows you to convert a shape from one shape type to another. When you convert a path shape that contains curves to a polygon shape, QuickDraw GX uses the curve error of the shape’s style to determine how close to make the polygon approximation. The distance between the polygon and the original path is never greater than the number of grid points (1/72-inch units) specified by the curve error.

Listing 3-5 shows a sample function that creates a circular path shape, sets its curve error to 1, and converts it to a polygon shape.

---

**Listing 3-5**      Converting a circle to a polygon

```
void ConvertCircleToPolygon(void)
{
    gxRectangle circleBounds = {ff(50), ff(50),
                                ff(200), ff(200)};

    gxShape    aCircle;

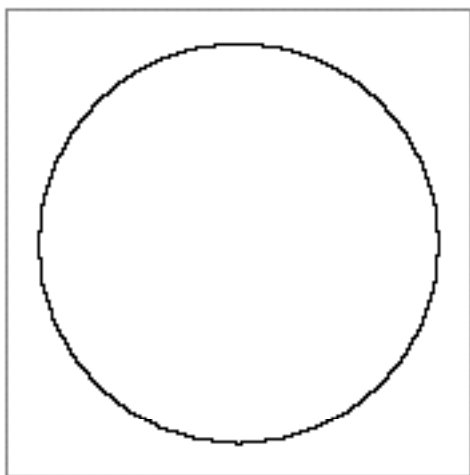
    aCircle = NewArc(&circleBounds, ff(0), ff(360), false);
    GXSetShapeFill(aCircle, gxClosedFrameFill);

    GXSetShapeCurveError(aCircle, ff(1));
    GXSetShapeType(aCircle, gxPolygonType);
    GXDrawShape(aCircle);
    GXDisposeShape(aCircle);
}
```

Since the curve error is 1 in this example, the resulting polygon is never more than 1 grid point away from the original circle, which makes for an accurate approximation, as shown in Figure 3-39.

---

**Figure 3-39**      Polygon approximation of a circle with curve error of 1

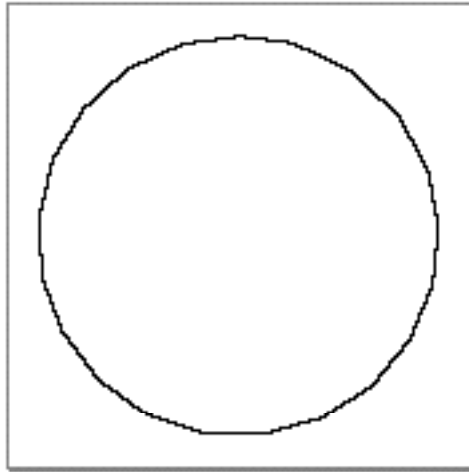




Increasing the curve error decreases the accuracy of the approximation. Setting the curve error to 5 in this example creates the polygon shown in Figure 3-40, which has fewer sides than the polygon in Figure 3-39.

---

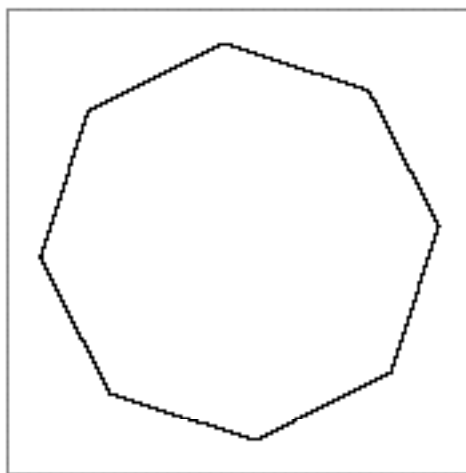
**Figure 3-40** Polygon approximation of a circle with curve error of 5



If you increase the curve error to 10, the octagon shown in Figure 3-41 results.

---

**Figure 3-41** Polygon approximation of a circle with curve error of 10

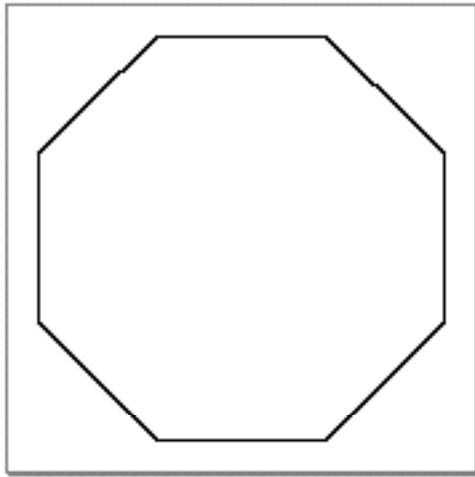


Although decreasing the curve error leads to more accurate approximations in general, a curve error of 0 is a special case. A curve error of 0 indicates that QuickDraw GX should not approximate the path at all. Instead, QuickDraw GX simply removes all off-curve control points, leaving a polygon made up of the on-curve geometric points of the initial path.

In Listing 3-5, the circular path returned by the `NewArc` library routine contains eight off-curve control points, which imply eight on-curve geometric points midway between each pair of off-curve control points. A curve error of 0 results in a polygon containing these eight on-curve points, as shown in Figure 3-42.

---

**Figure 3-42** Polygon resulting from a curve error of 0



For more information about paths, polygons, and on-curve and off-curve geometric points, see Chapter 3, “Geometric Shapes.”

For more information about curve error and the functions you can use to manipulate it, see “Curve Error” on page 3-14 and “Getting and Setting Curve Error” on page 3-114.

## Using Curve Error When Reducing Shapes

---

You can also use curve error to eliminate excess detail in complicated shapes. When you call the `GXReduceShape` or `GXSimplifyShape` functions, QuickDraw GX averages points within a curve error of each other.

You can use this feature to smooth a complicated contour, such as the wavy line created in Listing 3-6.

---

**Listing 3-6**      Creating a complicated contour

```
void FlattenWavyLine(void)
{
    gxShape aWave;

    static longwavyGeometry[] = {1, /* number of contours */
                                  13, /* number of points */
                                  0x2AA00000, /* 0010 0101 0101 */
                                  ff(80), ff(100), /* on */
                                  ff(110), ff(100), /* on */
                                  ff(113), ff(91), /* off */
                                  ff(118), ff(103), /* on */
                                  ff(123), ff(85), /* off */
                                  ff(128), ff(100), /* on */
                                  ff(133), ff(112), /* off */
                                  ff(135), ff(97), /* on */
                                  ff(141), ff(106), /* off */
                                  ff(145), ff(94), /* on */
                                  ff(150), ff(109), /* off */
                                  ff(153), ff(100), /* on */
                                  ff(183), ff(100) /* on */
    };

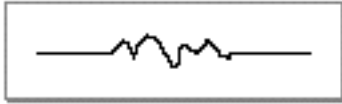
    aWave = GXNewPaths((gxPaths *) wavyGeometry);
    GXSetShapeFill(aWave, gxOpenFrameFill);

    GXDrawShape(aWave);

    GXDisposeShape(aWave);
}
```

The shape created by this sample function is shown in Figure 3-43.

**Figure 3-43** Wavy line



If you add the following lines of code to this sample function:

```
GXSetShapeCurveError(aWave, ff(10));  
GXReduceShape(aWave, 0);
```

the resulting shape has a slightly smoother appearance because QuickDraw GX averages sequential on-curve geometric points within the specified distance (a distance of 10 grid points). Figure 3-44 shows the resulting shape.

**Figure 3-44** Wavy line somewhat smoothed by curve error of 10



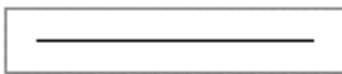
Increasing the curve error increases the number of geometric points that QuickDraw GX averages. A curve error of 15 results in the shape shown in Figure 3-45.

**Figure 3-45** Wavy line smoothed by curve error of 15



A curve error of 20 results in a completely straight line—all of the points between the start point and the end point of the contour have been averaged out as shown in Figure 3-46.

**Figure 3-46** Wavy line completely straightened by curve error of 20



**Note**

When QuickDraw GX reduces a shape, it does not ignore the first and last points of the contour. If these points had been close enough to the other points in this example, they, too, would have been averaged, and the entire shape would have become a point.  $\cup$

For more information about curve error and the functions you can use to manipulate it, see “Curve Error” on page 3-14 and “Getting and Setting Curve Error” beginning on page 3-114.

## Manipulating Pen Width and Placement

---

The pen width property of a style object determines the width with which QuickDraw GX draws a shape’s contours, and three of the style attributes determine where QuickDraw GX places the pen in relation to a shape’s contours. These three attributes are

- n the center-frame style attribute (`gxCenterFrameStyle`)
- n the inside-frame style attribute (`gxInsideFrameStyle`)
- n the outside-frame style attribute (`gxOutsideFrameStyle`)

Since contour direction and crossed contours affect pen placement, the examples in this section use a path shape shaped like a figure eight, as defined in Listing 3-7.

---

**Listing 3-7**     Defining a figure eight

```
void CreateFigureEight(void)
{
    gxShape aPathShape;

    static long figureEightGeometry[] = {1, /* number of contours */
                                         4, /* number of points */
                                         0xF0000000, /* 1111 ... */
                                         ff(50), ff(50),
                                         ff(200), ff(200),
                                         ff(50), ff(200),
                                         ff(200), ff(50)};

    aPathShape = GXNewPaths((gxPaths *) figureEightGeometry);
    GXSetShapeFill(aPathShape, gxClosedFrameFill);

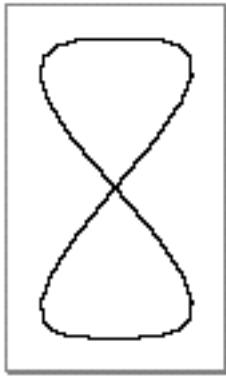
    GXDrawShape(aPathShape);

    GXDisposeShape(aPathShape);
}
```

Figure 3-47 shows the result of this sample function with the default pen width, which is a hairline, and the default pen placement, which is centered (as it always is for hairlines).

---

**Figure 3-47** A hairline figure eight



To increase the width of the pen, you can add the following line of code to the `CreateFigureEight` sample function:

```
GXSetShapePen(aPathShape, ff(30));
```

which results in the shape depicted in Figure 3-48.

---

**Figure 3-48** A thick figure eight



To change the placement of the thick pen, you can use the `GXSetShapeStyleAttributes` function to set the inside-frame style attribute or outside-frame style attribute. For example, if you add this line of code to the `CreateFigureEight` sample function:

```
GXSetShapeStyleAttributes(aPathShape, gxInsideFrameStyle);
```

QuickDraw GX shifts the entire pen width, which is 30 points, to the inside of the figure eight. Since, by default, QuickDraw GX defines the inside of a contour to be the right side, contour direction is significant in this case, and the resulting shape appears as depicted in Figure 3-49.

---

**Figure 3-49** A figure eight with pen inset



## Geometric Styles

If you indicate that the pen should be placed outside—that is, to the left of—the contour, using this line of code:

```
GXSetShapeStyleAttributes(aPathShape, gxOutsideFrameStyle);
```

the figure reverses, appearing as shown in Figure 3-50.

---

**Figure 3-50** A figure eight with pen outset



The contour direction of the path shape determines which side is the inside and which side is the outside. If you reverse the contour direction by reversing the order of the geometric points with this definition:

```
static long figureEightGeometry[] = {1, /* number of contours */
                                     4, /* number of points */
                                     0xF0000000, /* 1111 ... */
                                     ff(200), ff(50)
                                     ff(50), ff(200),
                                     ff(200), ff(200),
                                     ff(50), ff(50)};
```

but still set the outside-frame style attribute:

```
GXSetShapeStyleAttributes(aPathShape, gxOutsideFrameStyle);
```

then the resulting shape appears to be the same as the original figure-eight shape with the path *inset*, as shown in Figure 3-51.



**Figure 3-51** A reversed figure eight with pen outset

However, this figure still doesn't *look* like a figure eight with the path outset—it looks like a figure eight with the upper half of the path outset and the lower half of the path inset. This problem arises because the path crosses itself. To fix this problem, you can call the `GXSimplifyShape` function, which redefines the geometry of the shape so that the path has no crossed contours. Listing 3-8 shows a sample function that uses the `GXSimplifyShape` to remove the unwanted contour crossing.

**Listing 3-8** Removing unwanted contour crossings

```
void CreateUncrossedFigureEight(void)
{
    gxShape aPathShape;

    static long figureEightGeometry[] = {1, /* number of contours */
                                          4, /* number of points */
                                          0xF0000000, /* 1111 ... */
                                          ff(50),  ff(50), /* off */
                                          ff(200), ff(200), /* off */
                                          ff(50),  ff(200), /* off */
                                          ff(200), ff(50)}; /* off */

    aPathShape = GXNewPaths((gxPaths *) figureEightGeometry);
    GXSetShapeFill(aPathShape, gxClosedFrameFill);
    GXSetShapePen(aPathShape, ff(30));
    GXSimplifyShape(aPathShape);
}
```

## Geometric Styles

```

GXSetShapeStyleAttributes(aPathShape, gxAutoInsetStyle);
GXSetShapeStyleAttributes(aPathShape, gxOutsideFrameStyle);

GXDrawShape(aPathShape);

GXDisposeShape(aPathShape);
}

```

This sample function calls `GXSimplifyShape` to uncross the contours of the figure eight. However, you cannot be sure whether `GXSimplifyShape` uncrosses the contours by reversing the direction of the upper half of the figure eight, making each loop clockwise, or by reversing the lower half of the figure eight, making each loop counterclockwise. Therefore, the `CreateUncrossedFigureEight` sample function sets the auto-inset style attribute, which overrides the default assumption that the right side of the contour is the inside. Instead, QuickDraw GX determines the true inside of each contour.

Finally, now that the contours of the path do not cross and QuickDraw GX is determining the actual inside of the contour, setting the outside-frame style attribute works more as you would expect, as shown in Figure 3-52.

---

**Figure 3-52**      Uncrossed figure eight with pen outset



For more information about pen placement, see “Pen Placement” beginning on page 3-18. For more information about style attributes, see “Style Attributes” beginning on page 3-17 and “Style Attributes” beginning on page 3-98.

## Adding Caps to a Shape

---

To add a cap shape to the ends of another shape's contours, you must create a cap record structure. The cap structure has three fields: one for the start cap shape, one for the end cap shape, and one for the cap attributes.

Listing 3-9 shows how to create a cap structure with an arrow head for the start cap, an arrow tail for the end cap, and no cap attributes.

---

**Listing 3-9**      Creating an arrow

```
void CreateArrow(void)
{
    gxShape aCurve, arrowHead, arrowTail;

    static gxCurve curveGeometry = {ff(25), ff(125),
                                     ff(100), 0,
                                     ff(225), ff(125)};

    static long arrowHeadPolygonGeometry[] = {4, /* # of points */
                                              -ff(3), 0,
                                              0, fixed1,
                                              fixed1, 0,
                                              0, -fixed1};

    static long arrowTailPolygonGeometry[] = {5, /* # of points */
                                              -fixed1, 0,
                                              0, fixed1,
                                              ff(2), fixed1,
                                              ff(2), -fixed1,
                                              0, -fixed1};

    gxCapRecord theCapRecord;

    aCurve = GXNewCurve (&curveGeometry);

    arrowHead = NewPolygon((gxPolygon *)
                           &arrowHeadPolygonGeometry);
    arrowTail = NewPolygon((gxPolygon *)
                           &arrowTailPolygonGeometry);

    theCapRecord.startCap = arrowHead;
    theCapRecord.endCap = arrowTail;
    theCapRecord.attributes = gxNoAttributes;
}
```

## Geometric Styles

```

    GXSetShapeCap(aCurve, &theCapRecord);

    GXDisposeShape(arrowHead);
    GXDisposeShape(arrowTail);

    GXSetShapePen(aCurve, ff(10));

    GXDrawShape(aCurve);

    GXDisposeShape(aCurve);
}

```

This sample function creates two polygon shapes: one for the arrow head and one for the arrow tail. It then creates a cap structure that contains references to the two shape objects and an attributes field with no attributes set.

The sample function then calls the `GXSetShapeCap` function, which sets the cap property of the curve shape's style object. (Remember, it makes a copy of this style object if the style is shared amongst multiple shapes.)

When the `GXSetShapeCap` function copies the start cap and the end cap from the cap record to the curve's style object, it does not simply copy references to the arrow head polygon and the arrow tail polygon. Instead, it makes copies of those shapes and includes the copies in the cap property of the curve's style object. After setting the curve shape's caps, you may subsequently make changes to the arrow head and arrow tail shapes without affecting the start cap or end cap of the curve.

**Note**

Actually, the `GXSetShapeCap` function does not copy the entire start cap shape or end cap shape. Instead, it copies only the geometric information of the start and end cap shapes. For this reason, you must provide start cap shapes and end cap shapes in their primitive forms. <sup>u</sup>

After the `CreateArrow` sample function sets the caps of the curve shape, it disposes of the arrow head and arrow tail polygons. At this point, the owner count of these shapes becomes 0 (since the curve's style does not actually reference these shapes), and the memory used by the two polygon shapes is freed.

**Note**

In the same way that the `GXSetShapeCap` function copies geometry information from the start and end cap shapes into a style's cap property, the `GXGetShapeCap` function creates new shape objects and copies the geometry information from a style's cap property into the new shapes. If you use the `GXGetShapeCap` function to find the caps of a shape and alter those caps, you must use the `GXSetShapeCap` function to copy your changes back into the shape's caps. <sup>u</sup>

Figure 3-53 shows the result of the `CreateArrow` sample function.

**Figure 3-53** An arrow



Notice that QuickDraw GX rotates the start cap and the end cap to match the slope of the curve's contour, and scales them by the width of the pen. You can suppress the rotation by setting the level start-cap attribute and the level end-cap attribute.

The sections "The Cap Structure" on page 3-99 and "Cap Attributes" on page 3-101 describe the cap structure and the cap attributes in more detail, and the section "Getting and Setting Caps" beginning on page 3-123 describes the functions you can use to manipulate caps.

## Adding Standard Caps to a Shape

Two types of caps that you may frequently want to add to your shapes are the round cap and the square cap. The sample function in Listing 3-10 shows how to create these types of caps.

For a round cap, you need to create a semicircle, which you can do using the library function `NewArc`. To fit the end of a contour, the bounds of this semicircle must be set as follows:

```
gxRectangle roundCapBounds = {-fl(.5), -fl(.5), fl(.5), fl(.5)};
```

and the semicircle must start at 180 degrees and span a 180 degree arc:

```
gxRoundCap = NewArc(&roundCapBounds, ff(180), ff(180), true);
```

For a square cap to fit the end of a contour, its bounds must be set as follows:

```
gxRectangle squareCapBounds = {-ff(.5), -ff(.5), ff(0), ff(.5)};
```

Listing 3-10 shows how to create a round cap and a square cap for the curve shape from previous examples.

---

**Listing 3-10** Adding round caps and square caps to a curve

```
void CreateMyShape(void)
{
    gxShape  aCurve, gxRoundCap, gxSquareCap;

    static gxCurve curveGeometry = {ff(25),  ff(125),
                                     ff(100), 0,
                                     ff(225), ff(125)};

    static gxRectangle roundCapBounds = {-fl(.5), -fl(.5),
                                          fl(.5), fl(.5)};

    static gxRectangle squareCapBounds = {-ff(.5), -ff(.5),
                                           ff(0), ff(.5)};

    gxCapRecord theCapRecord;

    aCurve = GXNewCurve (&curveGeometry);

    gxRoundCap = NewArc(&roundCapBounds, ff(180), ff(180), true);
    gxSquareCap = GXNewRectangle(&squareCapBounds);

    theCapRecord.startCap = gxRoundCap;
    theCapRecord.endCap = gxSquareCap;
    theCapRecord.attributes = gxNoAttributes;

    GXSetShapeCap(aCurve, &theCapRecord);

    GXDisposeShape(gxRoundCap);
    GXDisposeShape(gxSquareCap);

    GXSetShapePen(aCurve, ff(10));

    GXDrawShape(aCurve);

    GXDisposeShape(aCurve);
}
```

Figure 3-54 shows the result of this sample function.

**Figure 3-54** Round and square caps



Notice that QuickDraw GX rotates and resizes the caps to fit the contour.

The sections “The Cap Structure” on page 3-99 and “Cap Attributes” on page 3-101 describe the cap structure and the cap attributes in more detail, and the section “Getting and Setting Caps” beginning on page 3-123 describes the functions you can use to manipulate caps.

## Adding Joins to a Shape

To add a join shape to the corners of another shape’s contours, you must create a join structure. The join structure has three fields: one for the join shape, one for the join attributes, and one for the miter, which is used only for sharp joins.

Listing 3-11 shows how to create a join structure with an diamond shape as the join shape, and then apply the diamond join shape to the corners of a rectangle shape.

**Listing 3-11** Adding joins to a shape

```
void CreateJoinedSquare(void)
{
    gxShape    aSquareShape, aDiamondShape;

    static gxRectangle squareGeometry = {ff(50), ff(50),
                                         ff(150), ff(150)};

    static long diamondGeometry[] = {1, /* number of contours */
                                     4, /* number of points */
                                     ff(0), ff(3),
                                     ff(1), fl(0),
                                     ff(0), -ff(3),
                                     -ff(1), ff(0)};
```

## Geometric Styles

```

    gxJoinRecord theJoinRecord;

    aSquareShape = GXNewRectangle(&squareGeometry);
    GXSetShapeFill(aSquareShape, gxClosedFrameFill);

    aDiamondShape = GXNewPolygons((gxPolygons *) diamondGeometry);

    theJoinRecord.attributes = gxNoAttributes;
    theJoinRecord.join = aDiamondShape;
    theJoinRecord.miter = 0;

    GXSetShapeJoin(aSquareShape, &theJoinRecord);

    GXDisposeShape(aDiamondShape);

    GXSetShapePen(aSquareShape, ff(10));

    GXDrawShape(aSquareShape);

    GXDisposeShape(aSquareShape);
}

```

This sample function creates a square as the shape to add joins to and a diamond-shaped polygon to use for the joins. It then creates a join structure which contains a reference to the diamond shape, an attributes field with no attributes set, and a miter of 0.

The sample function then calls the `GXSetShapeJoin` function, which sets the join property of the square shape's style object. (Remember, it makes a copy of this style object if the style is shared amongst multiple shapes.)

**Note**

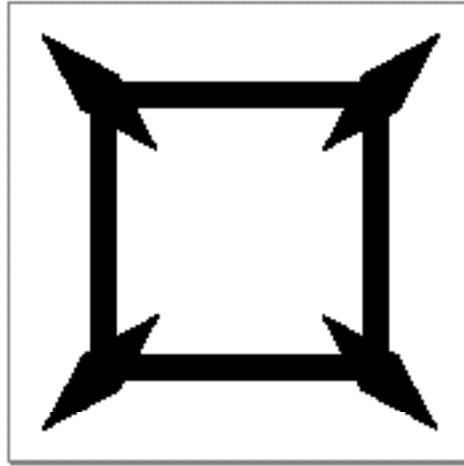
As with caps, QuickDraw GX copies only the geometric information of the join shape into the join property of the style object; it does not copy the entire join shape. For this reason, join shapes must be in their primitive form. Once you have called `GXSetShapeJoin`, you are free to change the original join shape without affecting the joins that you have already added to a shape. <sup>u</sup>

After the `CreateJoinedSquare` sample function sets the joins of the square shape, it disposes of the diamond-shaped polygon. At this point, the owner count of this polygon shape becomes 0 and the memory used by the polygon shape is freed.

Figure 3-55 shows the result of the `CreateJoinedSquare` sample function.



---

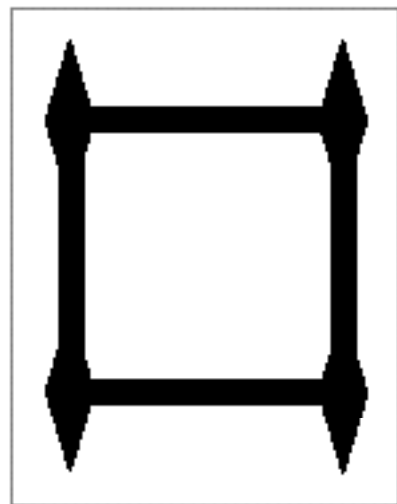
**Figure 3-55** A square with diamond-shaped joins

Notice that QuickDraw GX scales the join shape by the pen width and rotates the join shape to match the mid-angle of the two line segments that make each corner. You can suppress the rotation by setting the level join attribute:

```
theJoinRecord.attributes = gxLevelJoin;
```

Figure 3-56 shows the result of setting this attribute.

---

**Figure 3-56** A square with level joins

The sections “The Join Structure” on page 3-101 and “Join Attributes” on page 3-102 describe the join structure and join attributes in more detail, and the section “Getting and Setting Joins” beginning on page 3-129 describes the functions you can use to manipulate joins.

The next section shows how to create standard joins and how to use the miter field of the join structure.

## Adding Standard Joins to a Shape

---

Two types of joins that you may frequently want to add to your shapes are the round join and the square join. Unlike the standard cap shapes, which you add yourself by creating a semicircle shape or a half-square shape, the standard join shapes are provided for you by QuickDraw GX.

To create a standard join shape, you set the `join` field of the join record to `nil`, which indicates that you are not providing a join shape, and you set the sharp join attribute or the curve join attribute, which indicates that you want QuickDraw GX to generate one of the standard joins for you.

Listing 3-12 shows how to add a sharp join to an angle shape.

---

**Listing 3-12** Adding a sharp join to an angle shape

```
void CreateSharpJoin(void)
{
    gxShape  anAngleShape;

    static long angleGeometry[] = {1, /* number of contours */
                                   3, /* number of points */
                                   ff(20), ff(20),
                                   ff(250), ff(60),
                                   ff(20), ff(100)};

    gxJoinRecord theJoinRecord;

    anAngleShape = GXNewPolygons((gxPolygons *) angleGeometry);
    GXSetShapeFill(anAngleShape, gxOpenFrameFill);

    theJoinRecord.attributes = gxSharpJoin;
    theJoinRecord.join = nil;
    theJoinRecord.miter = gxPositiveInfinity;

    GXSetShapeJoin(anAngleShape, &theJoinRecord);

    GXSetShapePen(anAngleShape, ff(15));
}
```

```

GXDrawShape(anAngleShape);

GXDisposeShape(anAngleShape);
}

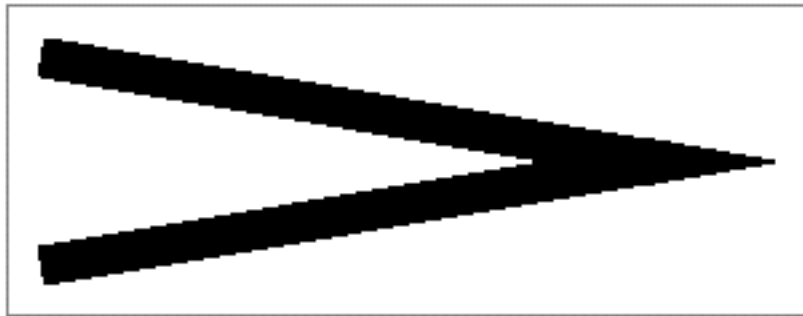
```

Notice that this sample function sets the `miter` field to the constant value `gxPositiveInfinity`, which indicates the join should be as sharp as necessary.

Figure 3-57 shows the result of this sample function.

---

**Figure 3-57** An angle with a sharp join



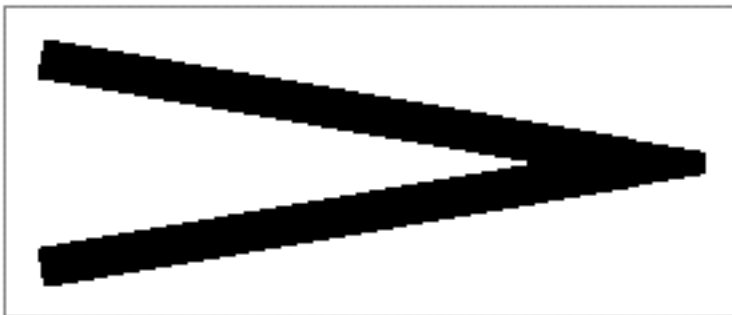
If you limit the miter of the sharp join, for example, with the code

```
theJoinRecord.miter = ff(1); /* scaled by pen width */
```

QuickDraw GX limits the distance between the actual corner of the contour as specified in the shape's geometry and the tip of the corner as actually drawn. Since miter is scaled by pen width, and the pen width in this example is 15, QuickDraw GX truncates the sharp join 15 points away from the actual corner of the geometry, as shown in Figure 3-58.

---

**Figure 3-58** An angle with a truncated sharp join



The sections “The Join Structure” on page 3-101 and “Join Attributes” on page 3-102 describe the join record structure and the join attributes in more detail, and the section “Getting and Setting Joins” beginning on page 3-129 describes the functions you can use to manipulate joins.

## Dashing a Shape

---

To add a dash shape along the contours of another shape, you must create a dash structure. The dash structure has five fields:

- n the dash attributes, which modify the way the shape is dashed
- n the dash shape, which contains the shape to use as the dashes
- n the dash advance, which determines the number of points between the start of one dash and the start of the next
- n the dash phase, which determines how far into the advance the dashing should start
- n the dash scale, which you can use to counteract the automatic scaling of the dash shape

The sample function in Listing 3-13 creates a curve shape dashed with diamonds. First, it creates the curve shape and the diamond shape. The diamond shape has a height and a width of 30.0 points.

The sample function then creates a dash structure for the diamond dashes, and calls the `GXSetShapeDash` function to set the dash property of the curve shape’s style object.

---

**Listing 3-13** Creating a curve shape dashed with diamonds

```
void CreateDashedCurve(void)
{
    gxShape  aCurveShape, aDiamondShape;

    static gxCurve curveGeometry = {ff(50), ff(125),
                                     ff(125), 0,
                                     ff(250), ff(125)};

    static long diamondGeometry[] = {1, /* number of contours */
                                     4, /* number of points */
                                     ff(0), ff(15),
                                     ff(15), fl(0),
                                     ff(0), -ff(15),
                                     -ff(15), ff(0)};

    gxDashRecord theDashRecord;
```

## Geometric Styles

```

aCurveShape = GXNewCurve (&curveGeometry);

aDiamondShape = GXNewPolygons((gxPolygons *) diamondGeometry);

theDashRecord.attributes = gxNoAttributes;
theDashRecord.dash = aDiamondShape;
theDashRecord.advance = ff(40);
theDashRecord.phase = 0;
theDashRecord.scale = ff(30);

GXSetShapeDash(aCurveShape, &theDashRecord);

GXDisposeShape(aDiamondShape);

GXSetShapePen(aCurveShape, ff(30));

GXDrawShape(aCurveShape);

GXDisposeShape(aCurveShape);
}

```

**Note**

As with caps and joins, QuickDraw GX copies only the geometric information of the dash shape into the dash property of the style object; it does not copy the entire dash shape. For this reason, the dash shape must be in its primitive form. Once you have called `GXSetShapeDash`, you are free to change the original dash shape without affecting the dashes of the dashed shape. <sup>u</sup>

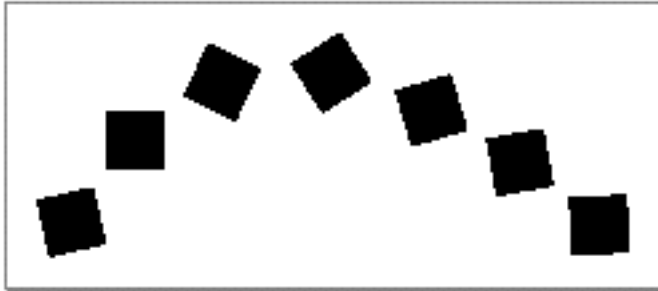
Notice that this sample function sets the dash advance to 40. Since the diamond shape is 30 points wide, this dash advance allows for 10 points between dashes. The dash phase is set to 0, which indicates that the origin of the first dash should be aligned with the beginning of the curve contour exactly.

Since QuickDraw GX scales dashes (perpendicularly to the dashed contour) by the pen width, the dashes in this example would be 900 points from tip to tip, as the diamond shape itself is 30 points high and the pen width of the curve is also 30 points. However, the sample function sets the dash scale to 30, by which QuickDraw GX scales the dashes down (again, perpendicularly to the dashed contour), which leaves the diamond shapes with their original size.

Figure 3-59 shows the result of the `CreateDashedCurve` sample function.

---

**Figure 3-59** A dashed curve



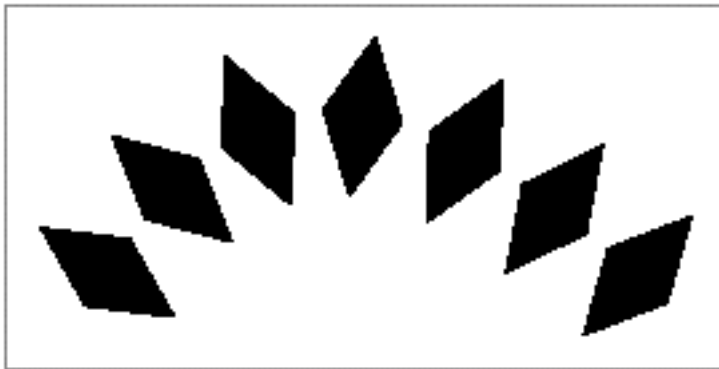
If you provide a smaller value for the dash scale, QuickDraw GX scales the dashes up in the direction perpendicular to the dashed contour. For example, if you provide a dash scale half as large:

```
theDashRecord.scale = ff(15);
```

the dashes become twice the size in the direction perpendicular to the curve, as shown in Figure 3-60.

---

**Figure 3-60** A curve with scaled dashes



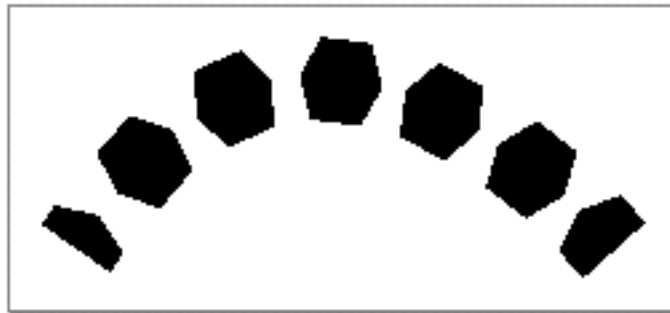
The dashes are now actually wider than the pen width of the curve. You can set the clip dash attribute to draw only the parts of the dashes that lie within the curve's pen width. For example, adding this line of code to the sample function:

```
theDashRecord.attributes = gxClipDash;
```

creates the shape shown in Figure 3-61.

---

**Figure 3-61** A curve with clipped dashes



Notice that QuickDraw GX not only clips the dashes to the width of the curve, but also clips them at the ends of the curve. To shift the dashes along the curve so that you see the whole first dash, you can adjust the dash phase. For example, this line of code:

```
theDashRecord.phase = GXFloatToFract(0.50);
```

shifts the dashes forward one half of the dash advance. Since the dash advance in this case is 40, the dashes are shifted forward 20 points, as shown in Figure 3-62.

---

**Figure 3-62** A curve with phased dashes



In this case, adjusting the dash phase is sufficient to cause a whole number of dashes to show. In other cases, you may have to use the auto-advance dash attribute, which is described in the next section.

The sections “The Dash Structure” on page 3-103 and “Dash Attributes” on page 3-105 describe the dash record and dash attributes in more detail, and the section “Getting and Setting Dashes” beginning on page 3-134 describes the functions you can use to manipulate dashes.

## Adjusting Dashes to Fit Contours

Sometimes the dash advance does not divide evenly into the length of a contour and the dashes don’t look quite right. QuickDraw GX provides the auto-advance dash attribute (`gxAutoAdvanceDash`) to handle this situation.

For example, the sample function in Listing 3-14 creates a circle dashed with kite-shaped diamonds. It does not use the auto-advance dash attribute.

**Listing 3-14** Creating a dashed circle

```
void CreateDashedCircle(void)
{
    gxShape  aCircleShape, aDiamondShape;

    static gxRectanglecircleBounds = {ff(50), ff(50),
                                       ff(180), ff(180)};

    static long diamondGeometry[] = {1, /* number of contours */
                                     4, /* number of points */
                                     ff(0), ff(20),
                                     ff(15), ff(0),
                                     ff(0), -ff(40),
                                     -ff(15), ff(0)};

    gxDashRecord theDashRecord;

    aCircleShape = NewArc(&circleBounds, ff(0), ff(360), false);
    GXSetShapeFill(aCircleShape, gxHollowFill);

    aDiamondShape = GXNewPolygons((gxPolygons *) diamondGeometry);

    theDashRecord.attributes = gxNoAttributes;
    theDashRecord.dash = aDiamondShape;
    theDashRecord.advance = ff(30);
}
```

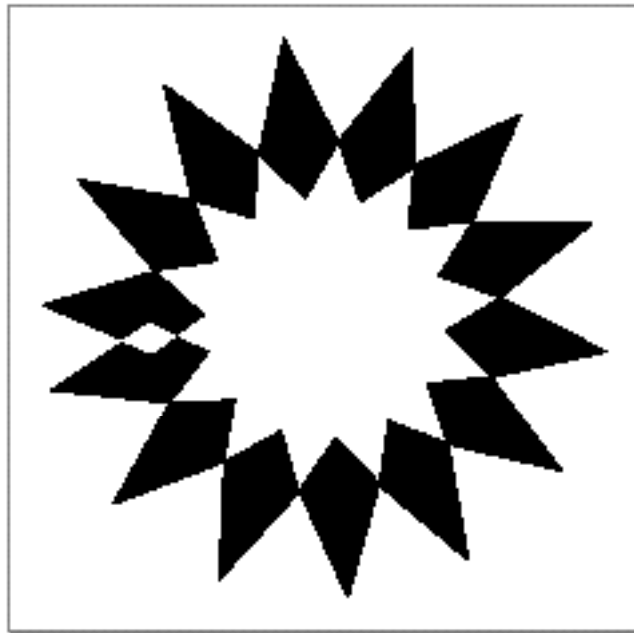


## Geometric Styles

```
theDashRecord.phase = 0;  
theDashRecord.scale = ff(60);  
  
GXSetShapeDash(aCircleShape, &theDashRecord);  
  
GXDisposeShape(aDiamondShape);  
  
GXSetShapePen(aCircleShape, ff(60));  
  
GXDrawShape(aCircleShape);  
}
```

Since this sample function does not set the auto-advance dash attribute, and the dash advance of 30 does not divide evenly into the circumference of the circle, this function results in the shape shown in Figure 3-63.

**Figure 3-63** Circle dashed with diamonds



Notice that the initial dash and the final dash overlap. (The overlapping region is not filled, because, by default, the dash shape has winding shape fill.)

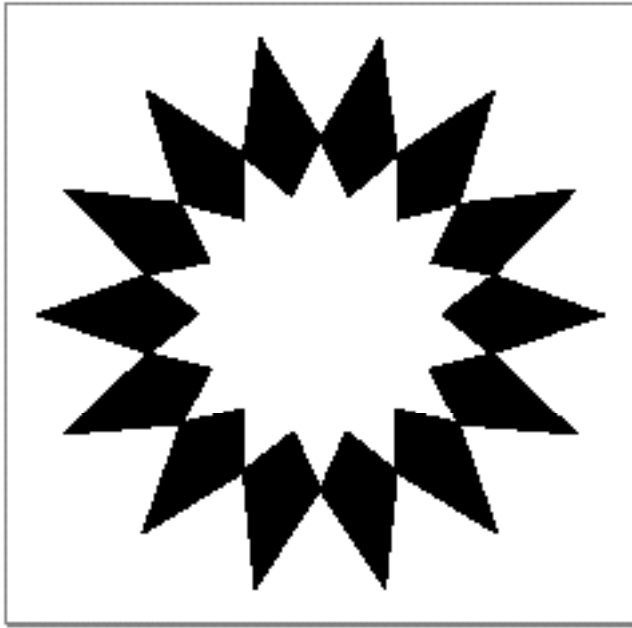
If, however, you set the auto-advance dash attribute, using this line of code:

```
theDashRecord.attributes = gxAutoAdvanceDash;
```

QuickDraw GX adjusts the dash advance accordingly. The result is shown in Figure 3-64.

---

**Figure 3-64** Circle with automatically advanced dashes



As you can see, QuickDraw GX adjusts the dash advance the smallest amount possible to create a whole number of dashes along the contour.

The sections “The Dash Structure” on page 3-103 and “Dash Attributes” on page 3-105 describe the dash structure and dash attributes in more detail, and the section “Getting and Setting Dashes” beginning on page 3-134 describes the functions you can use to manipulate dashes.

## Insetting Dashes

---

You can use a number of methods to change the placement of the dash shape relative to the dashed contour. For example, you can

- n set the inside-frame style attribute (`gxInsideFrameStyle`) or outside-frame style (`gxOutsideFrameStyle`) attribute of the style object containing the dash information so that QuickDraw GX places the dashes on the inside or outside of the contours
- n change the geometry of the dash shape so that QuickDraw GX changes the placement the dash shape correspondingly when dashing the shape

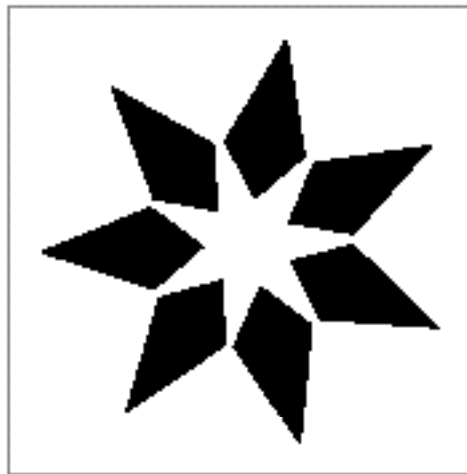
These two methods produce substantially different results. For example, if you inset the pen placement in the example from the previous section by adding the call

```
GXSetShapeStyleAttributes(aCircleShape, gxInsideFrameStyle);
```

to the `CreateADashedCircle` sample function in Listing 3-14 on page 3-70, QuickDraw GX automatically adjusts the number and spacing of the dashes to fit the smaller circle, as shown in Figure 3-65.

---

**Figure 3-65** Circle with diamond dashes inset

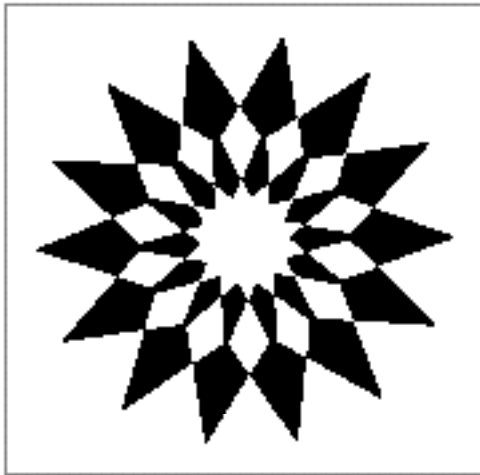


In this case, the number of dashes has been drastically reduced. If you want to keep the number of dashes constant, but move them towards the center of the circle, change the geometry of the dash shape instead of inseting the pen. For example, you can alter the diamond geometry from the `CreateDashedCircle` sample function by translating it up 30 points in the y-coordinate direction using this definition:

```
static long diamondGeometry[] = {1, /* number of contours */
                                4, /* number of points */
                                ff(0), ff(50),
                                ff(15), ff(30),
                                ff(0), -ff(10),
                                -ff(15), ff(30)};
```

In this case, if you do not inset the pen of the circle shape, the resulting shape maintains the greater number of dashes, but fits within the smaller circle, as shown in Figure 3-66.

**Figure 3-66** Circle with diamond dashes moved toward the center



The sections “The Dash Structure” on page 3-103 and “Dash Attributes” on page 3-105 describe the dash structure and dash attributes in more detail, and the section “Getting and Setting Dashes” beginning on page 3-134 describes the functions you can use to manipulate dashes.

## Breaking and Bending Dashes

You can use polygon shapes and path shapes as dash shapes, which means you can have a dash shape that has multiple contours. The way that QuickDraw GX place dashes along a contour can cause dashes with multiple contours to appear quite a distance from the dashed contour. QuickDraw GX provides the break dash attribute (`gxBreakDash`) and the bend dash attribute (`gxBendDash`) to address this problem.

As an example, you can create a dash shape with two entirely separate contours: for example, two separate diamonds, as shown in Figure 3-67.

**Figure 3-67** Dash shape with two contours



When you use this shape to dash any sort of curve, the larger diamond falls entirely off of the contour. Listing 3-15 creates a circle shape and dashes with the double diamond shape.

**Listing 3-15** Creating a dash with multiple contours

```
void CreateDoubleDiamondDash(void)
{
    gxShape    aCircleShape, aDiamondShape;

    gxRectangle circleBounds = {ff(50), ff(50), ff(180), ff(180)};

    static long doubleDiamond[] = {2, /* number of contours */
                                    4, /* number of points */
                                    ff(0), ff(10),
                                    ff(10), ff(0),
                                    ff(0), -ff(10),
                                    -ff(10), ff(0),
                                    4, /* number of points */
                                    ff(40), ff(10),
                                    ff(60), ff(0),
                                    ff(40), -ff(10),
                                    ff(20), ff(0)};

    gxDashRecord theDashRecord;

    aCircleShape = NewArc(&circleBounds, ff(0), ff(360), false);
    GXSetShapeFill(aCircleShape, gxClosedFrameFill);

    aDiamondShape = GXNewPolygons((gxPolygons *) doubleDiamond);
```

## Geometric Styles

```

    theDashRecord.attributes = gxAutoAdvanceDash;
    theDashRecord.dash = aDiamondShape;
    theDashRecord.advance = ff(80);
    theDashRecord.phase = GXFloatToFract(0.0);
    theDashRecord.scale = ff(60);

    GXSetShapeDash(aCircleShape, &theDashRecord);

    GXDisposeShape(aDiamondShape);

    GXSetShapePen(aCircleShape, ff(60));

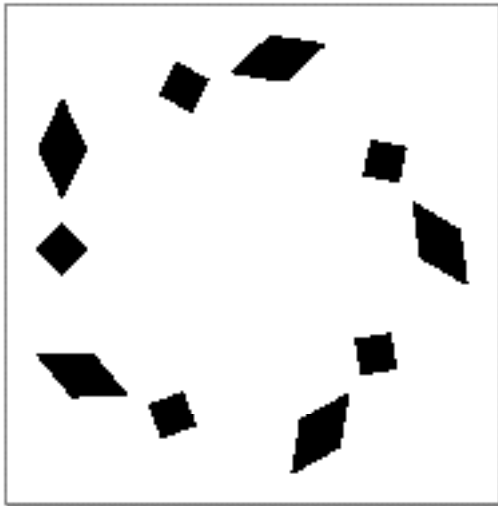
    GXDrawShape(aCircleShape);

    GXDisposeShape(aCircleShape);
}

```

This sample function creates the shape depicted in Figure 3-68.

**Figure 3-68** Circle dashed with double diamonds



The break dash attribute indicates that each contour of the dash shape should be separately rotated and placed on the contours of the dashed shape. If you set the break dash attribute in this example by replacing this line of code in the sample function:

```
theDashRecord.attributes = gxAutoAdvanceDash;
```

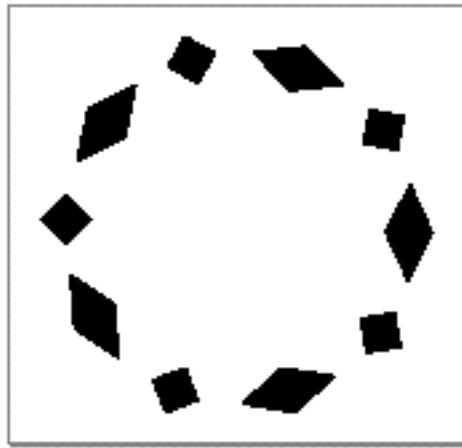
with this line of code:

```
theDashRecord.attributes = gxAutoAdvanceDash | gxBreakDash;
```

the resulting shape appears as shown in Figure 3-69.

---

**Figure 3-69** Circle with dashes broken



In this case, QuickDraw GX rotates and centers the large diamond contours (separately from the small diamond contours) to fit the contour of the dashed shape.

## Geometric Styles

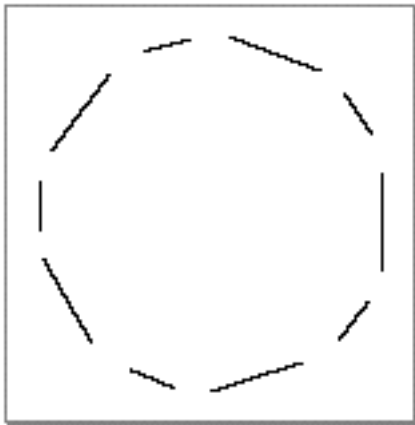
If you change the pen width of the circle in this example to 0.0, you get a hairline curve, and the dashes are mapped down to their one-dimensional image. So, for example, setting the pen width with the call

```
GXSetShapePen(aCircleShape, ff(0));
```

causes the dashed circle to appear as in Figure 3-70.

---

**Figure 3-70** Circle with hairline dashes



QuickDraw GX provides an extra feature for hairline dashes: you can bend them to fit curved contours exactly using the bend dash attribute (`gxBendDash`).



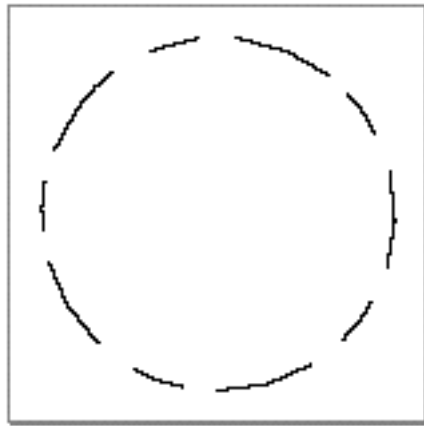
For example, if you change the dash attributes in this example using the assignment

```
theDashRecord.attributes = gxAutoAdvanceDash | gxBreakDash |  
                           gxBendDash;
```

the dashed circle appears as shown in Figure 3-71.

---

**Figure 3-71** Circle with bent hairline dashes



Note that you can specify the bend dash attribute only for hairline contours with broken dashes.

The sections “The Dash Structure” on page 3-103 and “Dash Attributes” on page 3-105 describe the dash record structure and dash attributes in more detail, and the section “Getting and Setting Dashes” beginning on page 3-134 describes the functions you can use to manipulate dashes.

## Wrapping Text to a Contour

---

You can wrap text to a contour by using a typographic shape as the dash shape. Since dashes must always be primitive shapes, you must convert a text or layout shape to a glyph or path shape before using it as a dash shape.

The sample function in Listing 3-16 creates a text shape, sets its font and text size, converts it to a path shape, and uses it to dash a curve.

---

**Listing 3-16**    Wrapping text

```
void WrapText(void)
{
    gxShape    aCurveShape, aTextShape;

    static gxCurve curveGeometry = {ff(25), ff(125),
                                    ff(100), 0,
                                    ff(225), ff(125)};

    gxDashRecord theDashRecord;

    aCurveShape = GXNewCurve(&curveGeometry);
    GXSetShapeFill(aCurveShape, gxOpenFrameFill);

    aTextShape = GXNewText(13,
                           (unsigned char *) "QuickDraw™ GX",
                           nil);
    SetShapeCommonFont(aTextShape, timesFont);
    GXSetShapeTextSize(aTextShape, ff(35));
    GXSetShapeType(aTextShape, gxPathType);

    theDashRecord.attributes = gxBreakDash;
    theDashRecord.dash = aTextShape;
    theDashRecord.advance = ff(330);
    theDashRecord.phase = 0;
    theDashRecord.scale = ff(35);
```

## Geometric Styles

```

GXSetShapeDash(aCurveShape, &theDashRecord);
GXDisposeShape(aTextShape);

GXSetShapePen(aCurveShape, ff(35));

GXDrawShape(aCurveShape);

GXDisposeShape(aCurveShape);
}

```

This example sets the dash scale to equal the text size so that the glyphs do not become distorted by dash scaling.

The result of this function is depicted in Figure 3-72. Notice that QuickDraw GX rotates and places each glyph separately on the contour because the break dash attribute is set.

**Figure 3-72**      Wrapped text



*Inside Macintosh: QuickDraw GX Typography* contains more information about using typographic shapes.

## Determining Dash Positions

A restriction of the QuickDraw GX dashing architecture is that each dash must be the same shape. There may be a situation where you'd like to dash a contour and have the dashes change as they progress along the contour.

To help you create the appearance of a dashed contours where the dashes change, QuickDraw GX provides the `GXGetShapeDashPositions` function. This function returns a list of mappings that identify the position and rotation of each dash on a shape.

By placing shapes in a picture using this list of mappings, you can give the effect of a contour with changing dashes.

As an example, the sample functions in this section show you how to create a picture of a clock. The `CreateDashedCircle` sample function in Listing 3-17 creates a circle with 12 dashes, each of which appears where a number would appear on a clock.

---

**Listing 3-17**    Creating a circle with 12 dashes

```
void CreateDashedCircle(void)
{
    gxShape  aCircleShape, aSquareShape;

    static gxRectangle circleBounds = {ff(50), ff(50),
                                       ff(180), ff(180)};

    static gxRectangle squareBounds = {-ff(10), -ff(10),
                                       ff(10),  ff(10)};

    gxDashRecord theDashRecord;

    aCircleShape = NewArc(&circleBounds, ff(30), ff(350), false);
    GXSetShapeFill(aCircleShape, gxClosedFrameFill);
    GXSetShapePen(aCircleShape, ff(60));

    aSquareShape = GXNewRectangle(&squareBounds);
    GXSetShapeFill(aSquareShape, gxEvenOddFill);

    theDashRecord.attributes = gxAutoAdvanceDash | gxLevelDash;
    theDashRecord.dash = aSquareShape;
    theDashRecord.advance = ff(34);
    theDashRecord.phase = 0;
    theDashRecord.scale = ff(60);

    GXSetShapeDash(aCircleShape, &theDashRecord);
    GXDisposeShape(aSquareShape);

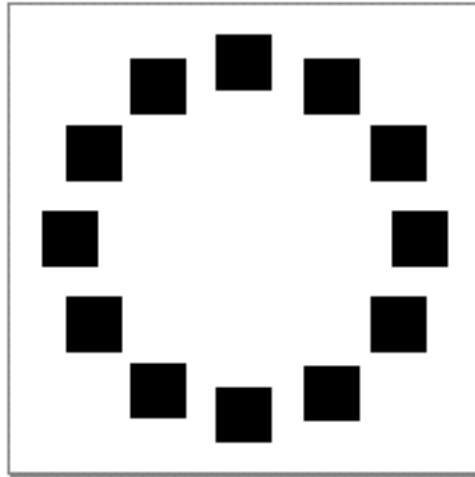
    GXDrawShape(aCircleShape);

    GXDisposeShape(aCircleShape);
}
```

This sample function creates a square shape using the `GXNewRectangle` function to use as a dash for a circle shape created using the library function `NewArc`.

The result of this function is shown in Figure 3-73.

**Figure 3-73** Dash positions for a clock



To replace the square dashes with numbers, the sample function in Listing 3-18 calls the `GetDashPositions` function to obtain an array of mappings that identify the position and rotation of each dash. (Notice that the dashes are not rotated in this case since the level dash attribute is set.)

The sample function in Listing 3-18 then creates a picture and adds to it text shapes containing the numbers 1 through 12. Each time text is added to the picture, its mapping is set to be the next mapping in the array of dash positions.

**Listing 3-18** Creating a clock shape

```
void CreateAClock(void)
{
    gxShape  aCircleShape, aTextShape, aSquareShape, aPicture;

    static gxRectangle circleBounds = {ff(50), ff(50),
                                       ff(180), ff(180)};

    static gxRectangle squareBounds = {-ff(10), -ff(10),
                                       ff(10),  ff(10)};

    static gxPointtextPosition = {ff(0), ff(0)};
```

## Geometric Styles

```

static char* numbers[] = { " 1", " 2", " 3", " 4", " 5", " 6",
                           " 7", " 8", " 9", "10", "11", "12"};

gxDashRecord theDashRecord;

long numberOfDashes, count;
gxMapping dashMappings[12];

/* Create the dashed circle from the previous example. */

aCircleShape = NewArc(&circleBounds, ff(30), ff(350), false);
GXSetShapeFill(aCircleShape, gxClosedFrameFill);

aSquareShape = GXNewRectangle(&squareBounds);
GXSetShapeFill(aSquareShape, gxEvenOddFill);

theDashRecord.attributes = gxAutoAdvanceDash | gxLevelDash;
theDashRecord.dash = aSquareShape;
theDashRecord.advance = ff(34);
theDashRecord.phase = GXFloatToFract(0.0);
theDashRecord.scale = ff(60);

GXSetShapeDash(aCircleShape, &theDashRecord);
GXSetShapePen(aCircleShape, ff(60));

/* Find the dash positions. */
numberOfDashes = GXGetShapeDashPositions(aCircleShape,
                                         dashMappings);

GXDisposeShape(aCircleShape);
GXDisposeShape(aSquareShape);

/* Create a picture with numbered text shapes. */

aTextShape = GXNewText(1, (unsigned char*) " 1",
                      &textPosition);
GXSetShapeFill(aTextShape, gxEvenOddFill);

aPicture = GXNewShape(gxPictureType);
GXSetShapeAttributes(aPicture, gxUniqueItemsShape);
for (count = 0; count <= numberOfDashes; count++) {
    GXSetShapeMapping(aTextShape, dashMappings[count]);
}

```

## Geometric Styles

```

        GXSetText(aTextShape, 2, numbers[count], &textPosition);
        AddToShape(aPicture, aTextShape);
    }
    GXDisposeShape(aTextShape);

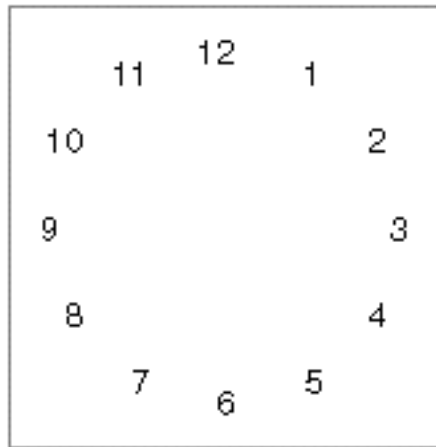
    GXDrawShape(aPicture);

    GXDisposeShape(aPicture);
}

```

The result of the `CreateAClock` sample function is depicted in Figure 3-74.

**Figure 3-74** A clock shape



This sample function uses some concepts from other parts of QuickDraw GX. For more information about

- n mappings, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.
- n pictures and adding elements to them, see Chapter 6, “Picture Shapes.”
- n typographic shapes, see *Inside Macintosh: QuickDraw GX Typography*

## Adding a Pattern to a Shape

---

To add a pattern to a shape, you must create a pattern structure. The pattern structure has four fields: the shape to use as the pattern, the pattern attributes, and a pair of vectors that define the grid over which QuickDraw GX places the pattern.

The sample function in Listing 3-19 creates a large rectangle shape patterned with small squares.

---

**Listing 3-19**     Patterning a shape

```
void CreatePatternedRectangle(void)
{
    gxShape    aRectangleShape, aSquarePattern;

    static gxRectangle rectangleGeometry = {ff(50), ff(50),
                                           ff(250), ff(150)};

    static gxRectangle squareGeometry = {ff(0), ff(0),
                                         ff(10), ff(10)};

    gxPatternRecord thePatternRecord;

    aRectangleShape = GXNewRectangle(&rectangleGeometry);

    aSquarePattern = GXNewRectangle(&squareGeometry);

    thePatternRecord.attributes = gxNoAttributes;
    thePatternRecord.pattern = aSquarePattern;
    thePatternRecord.u.x = ff(0);
    thePatternRecord.u.y = ff(20);
    thePatternRecord.v.x = ff(20);
    thePatternRecord.v.y = ff(0);

    GXSetShapePattern(aRectangleShape, &thePatternRecord);
    GXDisposeShape(aSquarePattern);

    GXDrawShape(aRectangleShape);

    GXDisposeShape(aRectangleShape);
}
```

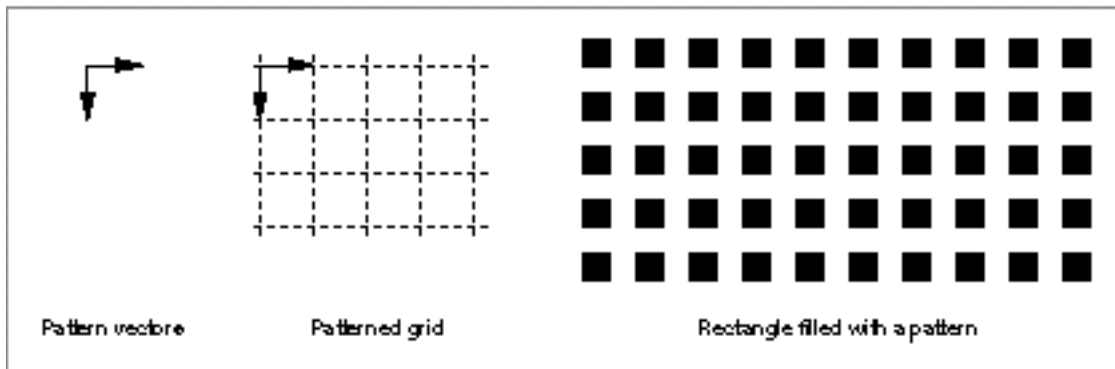


**Note**

As with caps, joins, and dashes, QuickDraw GX copies only the geometric information of the pattern shape into the pattern property of the style object; it does not copy the entire pattern shape. For this reason, pattern shapes must be in primitive form. Once you have called `GXSetShapePattern`, you are free to change the original pattern shape without affecting the pattern of the patterned shape.  $\cup$

Notice that this sample function creates a square pattern shape 10 points high by 10 points wide. It places that square pattern on a rectangular grid 20 points high by 20 points wide, resulting in the shape shown in Figure 3-75.

**Figure 3-75** A rectangle with a pattern



Although this example places the pattern shape on a rectangular grid, you are not limited to rectangular grids. The `u` and `v` fields of the pattern structure allow you to define a pair of vectors, so your pattern can be placed on any regular grid.

QuickDraw GX does not limit you to patterning filled shapes; you can pattern framed shapes as well. For example, if you change the previous example so that the rectangle shape is framed using the call

```
GXSetShapeFill(aRectangleShape, gxClosedFrameFill);
```

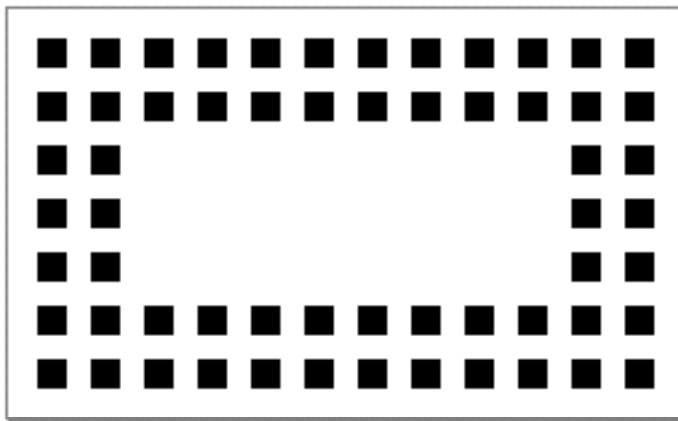
and has a thick pen width using the call

```
GXSetShapePen(aRectangleShape, ff(40));
```

the resulting function creates the shape shown in Figure 3-76.

---

**Figure 3-76** A framed rectangle with a pattern



You can also pattern dashed shapes. For examples, see “Combining Caps, Joins, Dashes, and Patterns” on page 3-91.

The sections “The Pattern Structure” on page 3-106 and “Pattern Attributes” on page 3-107 describe the pattern record structure and pattern attributes in more detail, and the section “Getting and Setting Patterns” beginning on page 3-142 describes the functions you can use to manipulate patterns.

## Determining Pattern Positions

---

As with the model for dashes, the QuickDraw GX model for patterns provides only for the case where the pattern shape remains the same throughout the entire patterned shape. If you want to pattern a shape and have the pattern change throughout it, you must use the `GXGetShapePatternPositions` function. This function returns an array of points that identify the location of each pattern shape on the patterned shape.

As an example, the sample function in this section shows you how to alter the patterned rectangle from the previous section. The sample function in Listing 3-20 first creates the patterned rectangle shown in Figure 3-75 and then uses the `GXGetShapePatternPositions` function to find the position of each small square in that patterned rectangle. The sample function then creates a picture, adding small squares at the appropriate positions, but rotating each new square a small amount.

---

**Listing 3-20** Changing a pattern throughout a patterned shape

```
void CreateBizarrePattern(void)
{
    gxShape aRectangleShape, smallRectangle, aPicture;

    static gxRectangle rectangleGeometry = {ff(50), ff(50),
                                           ff(250), ff(150)};

    static gxRectangle smallRectGeometry = {ff(0), ff(0),
                                           ff(10), ff(10)};

    gxPatternRecord thePatternRecord;

    gxPoint *patternPositions;

    int numberOfPatterns, count;

    aRectangleShape = GXNewRectangle(&rectangleGeometry);
    GXSetShapeFill(aRectangleShape, gxEvenOddFill);

    smallRectangle = GXNewRectangle(&smallRectGeometry);
    GXSetShapeFill(smallRectangle, gxEvenOddFill);

    thePatternRecord.attributes = gxPortAlignPattern;
    thePatternRecord.pattern = smallRectangle;
    thePatternRecord.u.x = ff(0);
    thePatternRecord.u.y = ff(20);
    thePatternRecord.v.x = ff(20);
    thePatternRecord.v.y = ff(0);

    GXSetShapePattern(aRectangleShape, &thePatternRecord);

    numberOfPatterns = GXGetShapePatternPositions(aRectangleShape,
                                                  nil);
}
```

## Geometric Styles

```

patternPositions = (gxPoint *)
                    NewPtr(numberOfPatterns * sizeof(gxPoint));
GXGetShapePatternPositions(aRectangleShape, patternPositions);

GXDisposeShape(aRectangleShape);

aPicture = GXNewShape(gxPictureType);
GXSetShapeAttributes(aPicture, gxUniqueItemsShape);
for (count = 0; count < numberOfPatterns; count++) {
    GXRotateShape(smallRectangle, ff(10), 0, 0);
    GXMoveShapeTo(smallRectangle, patternPositions[count].x,
                  patternPositions[count].y);
    AddToShape(aPicture, smallRectangle);
}
GXDisposeShape(smallRectangle);
DisposePtr((Ptr)patternPositions);

GXDrawShape(aPicture);

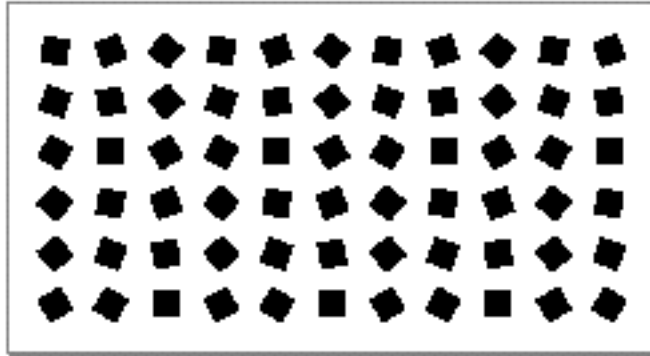
GXDisposeShape(aPicture);
}

```

This function calls the `GXGetShapePatternPositions` function twice. The first time, it sends `nil` as the value of the pattern positions array, which indicates that the `GXGetShapePatternPositions` function should not return an actual array of positions, but should return as the function result the total number of pattern positions. Once the sample function has this total, it allocates enough memory to hold the array of pattern positions, and then calls `GXGetShapePatternPositions` again to determine the actual positions.

The result of this sample function is shown in Figure 3-77.

**Figure 3-77** Shape with changing pattern



Notice that, in this case, the list of positions returned by `GXGetShapePatternPositions` starts at the upper-left corner and works down each column of the patterned shape. In general, the order of the positions returned by the `GXGetShapePatternPositions` function is not guaranteed by QuickDraw GX.

This sample function uses some concepts from other parts of QuickDraw GX. For more information about

- n rotating and moving shapes, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.
- n pictures and adding elements to them, see Chapter 6, “Picture Shapes,” in this book.

## Combining Caps, Joins, Dashes, and Patterns

As mentioned in “Interactions Between Caps, Joins, Dashes, and Patterns” on page 3-22, combining caps, joins, dashes, and patterns on the same shape causes some interesting interactions.

These elements interact differently in each of these three cases:

- n the shape does not have a dash but has one or more of the three other stylistic variations
- n the shape does have a dash but the clip dash attribute is not set
- n the shape does have a dash and the clip dash attribute is set

When a shape has a cap and a join, QuickDraw GX adds the caps to the beginnings and ends of the shape's contours, and adds the joins to the other on-curve geometric points of the shape's contours. If the shape also has a pattern, QuickDraw GX draws this pattern throughout the shape's frame as well as the shape's caps and joins. The sample function in Listing 3-21 creates an angle shape with a round cap, a square join, and a very small square pattern.

---

**Listing 3-21** Combining a cap, join, and pattern

```
void CapJoinPattern(void)
{
    gxShape    anAngleShape, aRoundCap, aSquareJoin, aSquarePattern;

    static long angleGeometry[] = {1, /* number of contours */
                                   3, /* number of points */
                                   ff(100), ff(100),
                                   ff(200), ff(80),
                                   ff(300), ff(100)};

    static long diamondGeometry[] = {1, /* number of contours */
                                      4, /* number of points */
                                      ff(0), ff(50),
                                      ff(10), ff(0),
                                      ff(0), -ff(50),
                                      -ff(10), ff(0)};

    static gxRectangle circleBounds = {-fl(.75), -fl(.75),
                                       fl(.75), fl(.75)};
    static gxRectangle smallSquareGeometry = {ff(0), ff(0),
                                              ff(1), ff(1)};

    gxCapRecord theCapRecord;
    gxJoinRecord theJoinRecord;
    gxPatternRecord thePatternRecord;

    /* Create the shape to be capped, joined, and patterned. */
    anAngleShape = GXNewPolygons((gxPolygons *) angleGeometry);
    GXSetShapeFill(anAngleShape, gxOpenFrameFill);
    GXSetShapePen(anAngleShape, ff(50));

    /* Create the round cap and add to the shape. */
    aRoundCap = NewArc(&circleBounds, ff(0), ff(360), false);
    theCapRecord.startCap = aRoundCap;
    theCapRecord.endCap = aRoundCap;
```

## Geometric Styles

```

theCapRecord.attributes = gxNoAttributes;
GXSetShapeCap(anAngleShape, &theCapRecord);
GXDisposeShape(aRoundCap);

/* Create the square join and add to join the shape. */
aSquareJoin = GXNewRectangle(&circleBounds);
theJoinRecord.attributes = gxNoAttributes;
theJoinRecord.join = aSquareJoin;
theJoinRecord.miter = 0;
GXSetShapeJoin(anAngleShape, &theJoinRecord);
GXDisposeShape(aSquareJoin);

/* Create the small square pattern and pattern the shape. */
aSquarePattern = GXNewRectangle(&smallSquareGeometry);
GXSetShapeFill(aSquarePattern, gxSolidFill);
thePatternRecord.attributes = gxNoAttributes;
thePatternRecord.pattern = aSquarePattern;
thePatternRecord.u.x = ff(0);
thePatternRecord.u.y = ff(2);
thePatternRecord.v.x = ff(2);
thePatternRecord.v.y = ff(0);
GXSetShapePattern(anAngleShape, &thePatternRecord);
GXDisposeShape(aSquarePattern);

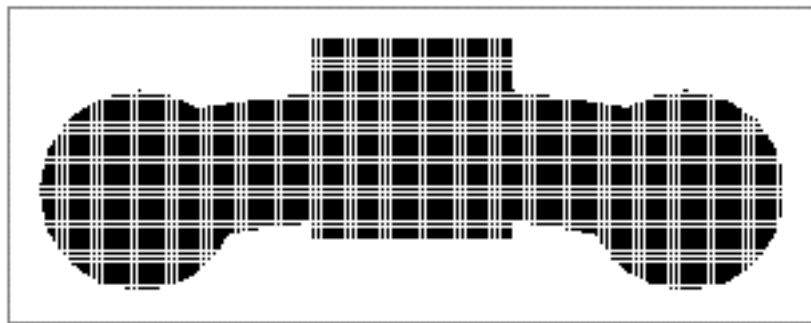
GXDrawShape(anAngleShape);

GXDisposeShape(anAngleShape);
}

```

The result of this function is shown in Figure 3-78.

**Figure 3-78** Angle shape with cap, join, and pattern



## Geometric Styles

The second case of cap, join, dash, and pattern interaction is when the shape has a dash but the clip dash attribute is not set. In this case, QuickDraw GX *ignores the caps and joins of the shape*. However, QuickDraw GX does draw the pattern throughout the dashes.

For example, if you add the following declarations at the appropriate places in the previous example:

```
gxShape aDiamondDash;

static long diamondGeometry[] = {1, /* number of contours */
                                4, /* number of points */
                                ff(0), ff(50),
                                ff(10), ff(0),
                                ff(0), -ff(50),
                                -ff(10), ff(0)};

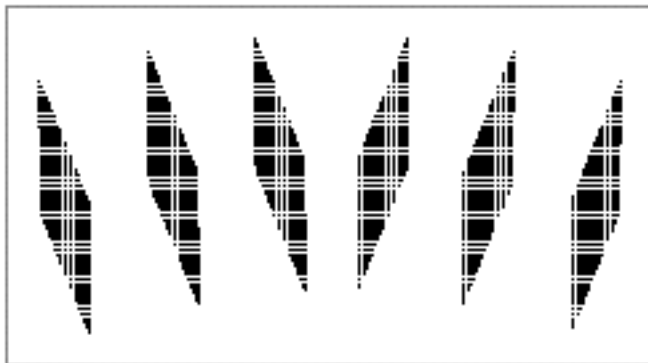
gxDashRecord theDashRecord;
```

and you add the following code to create a diamond-shaped dash:

```
/* Create the diamond dash and dash the shape. */
aDiamondDash = GXNewPolygons((gxPolygons *) diamondGeometry);
GXSetShapeFill(aDiamondDash, gxEvenOddFill);
theDashRecord.attributes = gxNoAttributes;
theDashRecord.dash = aDiamondDash;
theDashRecord.advance = ff(40);
theDashRecord.phase = 0;
theDashRecord.scale = ff(50);
GXSetShapeDash(anAngleShape, &theDashRecord);
GXDisposeShape(aDiamondDash);
```

the resulting shape will appear as depicted in Figure 3-79.

**Figure 3-79** Angle shape with dash and pattern; caps and join ignored





The third case of cap, join, dash, and pattern interaction is when the shape has a dash and the clip dash attribute is set. In this case, QuickDraw GX adds the cap and the join shapes to the clip shape used to clip the dashes. Patterns are not allowed in this case, so if you add the following line to the previous example:

```
theDashRecord.attributes = gxClipDash;
```

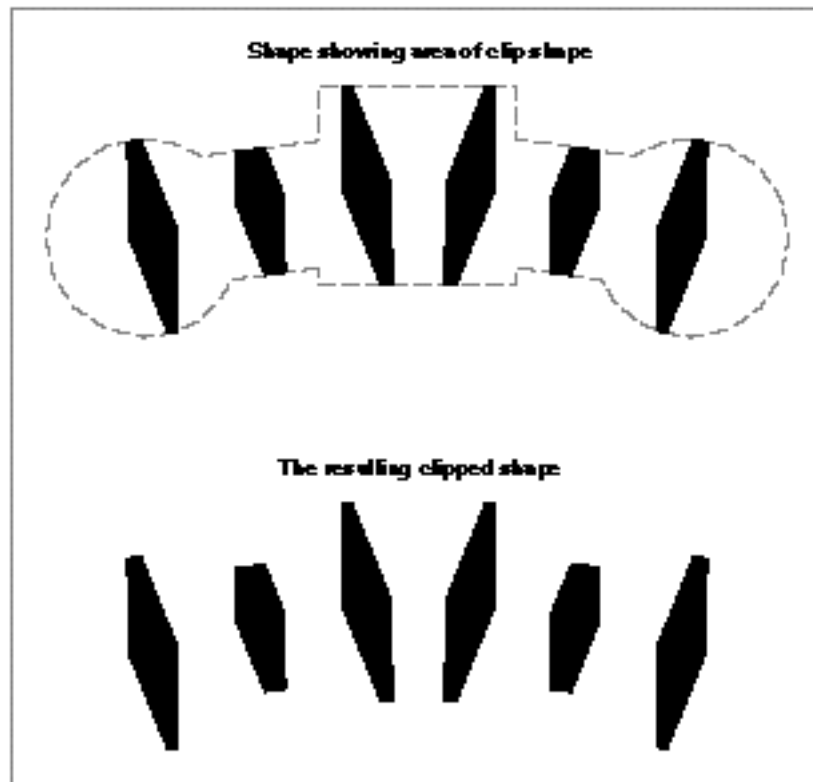
you must comment out this line:

```
/* GXSetShapePattern(anAngleShape, &thePatternRecord); */
```

which ensures that no pattern is set for the shape.

In this case, the resulting shape is drawn as shown in Figure 3-80.

**Figure 3-80** Shape with cap, join, dash, and the clip dash attribute set



Notice that the dashes (which are now solid because there is no pattern) are clipped to the thick contours of the angle shape. However, at the ends and at the corner more of the dashes show because the cap shapes and the join shape are added to the clip shape used to clip the dashes.

## Geometric Styles Reference

---

Each QuickDraw GX shape includes a shape object, a style object, an ink object, and a transform object. This section describes the data types and functions that are specific to style objects.

The “Constants and Data Types” section shows the type definition for the style object, and the structure and enumeration definitions used for five of the properties of style objects: the style attributes, the caps, the join, the dash, and the pattern.

The “Functions” section describes functions that manipulate the geometric style properties: the style attributes, the curve error, the pen width, the caps, the join, the dash, and the pattern. These properties allow you to apply stylistic variations to geometric shapes.

For information regarding creating and manipulating style objects themselves, or manipulating their tags and owner counts, see the chapter “Style Objects” in *Inside Macintosh: QuickDraw GX Objects*.

For information regarding the typographic style properties—for example, the font, text size, and text face—see the chapter “Typographic Styles” in *Inside Macintosh: QuickDraw GX Typography*.

### Constants and Data Types

---

This section describes the data types that you use to provide information about and to retrieve information from style objects.

You use the `gxStyle` data type when referring to a style object. This data type is described in full in the chapter “Style Objects” of *Inside Macintosh: QuickDraw GX Objects*.

You use the `gxStyleAttributes` enumeration when getting and setting individual flags of the attributes property of a style object.

You use the `gxCapRecord` structure and the `gxCapAttributes` enumeration when getting and setting the start cap and end cap of a shape.

You use the `gxJoinRecord` structure and the `gxJoinAttributes` enumeration when getting and setting the corner join of a shape.

You use the `gxDashRecord` structure and the `gxDashAttributes` enumeration when getting and setting a shape’s dashes.

You use the `gxPatternRecord` structure and the `gxPatternAttributes` enumeration when getting and setting a shape’s pattern.

## Style Objects

---

You use the `GxStyle` data type when referring to a style object. This data type is described in full in the chapter “Style Objects” of *Inside Macintosh: QuickDraw GX Objects*.

Style objects have owner counts, tags, typographic style properties, and seven geometric style properties. The owner count and tags properties are described in *Inside Macintosh: QuickDraw GX Objects*. The typographic style properties are described in *Inside Macintosh: QuickDraw GX Typography*. The geometric style properties are listed here:

- n **Style attributes.** This property is a group of flags that modify the behavior of the style object. The section “Style Attributes” on page 3-17 discusses the effects of these attributes. The section “Style Attributes” on page 3-98 describes the style attribute flags, and “Getting and Setting Style Attributes” on page 3-109 describes the functions you can use to examine or alter style attribute flags.
- n **Curve error.** This property specifies the allowable amount of error when QuickDraw GX converts a path shape into a polygon shape. It also specifies how far apart geometric points must be for QuickDraw GX to consider them separate points when reducing or simplifying a shape. The section “Curve Error” on page 3-14 discusses the curve error property and the sections “Using Curve Error When Converting Paths to Polygons” on page 3-45 and “Using Curve Error When Reducing Shapes” on page 3-49 give examples of using curve error. The section “Getting and Setting Curve Error” on page 3-114 describes the functions you can use to examine or alter this property.
- n **Pen width.** This property specifies the thickness of the pen QuickDraw GX uses to draw the contours of a shape. “The Geometric Pen” on page 3-15 describes how QuickDraw GX uses the pen when drawing, and “Getting and Setting the Pen Width” beginning on page 3-119 describes the functions you can use to examine or alter the pen width.
- n **Cap.** This property specifies what QuickDraw GX should draw at the start and the end of a shape’s contours. The section “Caps” on page 3-23 describes start and end caps, the sections “The Cap Structure” on page 3-99 and “Cap Attributes” on page 3-101 discuss the data types you use to describe start and end caps, and the section “Getting and Setting Caps” beginning on page 3-123 describes the functions you can use to examine or alter a shape’s start and end caps.
- n **Join.** This property specifies what QuickDraw GX draws at the corners of a shape’s geometry. The section “Joins” on page 3-25 describes corner joins, the sections “The Join Structure” on page 3-101 and “Join Attributes” on page 3-102 discuss the data types you use to describe corner joins, and the section “Getting and Setting Joins” beginning on page 3-129 describes the functions you can use to examine or alter a shape’s corner joins.

## Geometric Styles

- n **Dash.** This property specifies how QuickDraw GX should dash the contours of a shape. The section “Dashes” on page 3-27 describes dashes, the sections “The Dash Structure” on page 3-103 and “Dash Attributes” on page 3-105 discuss the data types you use to describe dashes, and the section “Getting and Setting Dashes” beginning on page 3-134 describes the functions you can use to examine or alter a shape’s dashes.
- n **Pattern.** This property specifies how QuickDraw GX should fill the geometry of a shape with a pattern. The section “Patterns” on page 3-31 describes patterns, the sections “The Pattern Structure” on page 3-106 and “Pattern Attributes” on page 3-107 discuss the data types you use to describe patterns, and the section “Getting and Setting Patterns” beginning on page 3-142 describes the functions you can use to examine or alter a shape’s pattern.

## Style Attributes

---

Each style object has a set of style attributes, which are a group of flags that modify the behavior of the style object. In particular, these flags allow you to specify how QuickDraw GX places the pen with respect to a shape’s geometry and whether the shape should be constrained to a grid. These constants are defined in the `gxStyleAttributes` enumeration:

```
enum gxStyleAttributes {
    gxCenterFrameStyle      = 0,
    gxSourceGridStyle       = 0x0001,
    gxDeviceGridStyle       = 0x0002,
    gxInsideFrameStyle      = 0x0004,
    gxOutsideFrameStyle     = 0x0008,
    gxAutoInsetStyle        = 0x0010
};
```

```
typedef long gxStyleAttribute;
```

### Constant descriptions

`gxCenterFrameStyle`

Indicates that QuickDraw GX should center the geometric pen along the shape’s contours.

`gxSourceGridStyle`

Constrains the geometric points of the shape in geometry space. When drawing a shape whose style object has this flag set, QuickDraw GX moves each geometric point of the shape’s geometry to the closest integral position before applying the shape’s style and transform. (Note that the original geometric points are unchanged; this operation occurs only as the shape is being drawn.) See “Grids” beginning on page 3-20 for more information.

## Geometric Styles

`gxDeviceGridStyle`

Constrains the geometric points of the shape in device space. When drawing a shape whose style object has this flag set, QuickDraw GX moves the shape's geometric points, *after* applying the shape's style and transform, to the closest integral position (that is, pixel position) in the device space. (Note that the original geometric points are unchanged; this operation occurs only as the shape is being drawn.) See “Grids” beginning on page 3-20 for more information.

`gxInsideFrameStyle`

Indicates that QuickDraw GX should position the pen along the inside of the shape's contours. By default, QuickDraw GX uses the direction of a contour to determine which side is the inside; the right side of a contour is considered the inside.

`gxOutsideFrameStyle`

Indicates that QuickDraw GX should place the pen along the outside of the shape's contours. By default, QuickDraw GX uses the direction of a contour to determine which side is the inside; the left side of a contour is considered the outside.

`gxAutoInsetStyle`

Alters the default definition of the inside and outside of a contour. When this flag is not set, QuickDraw GX assumes the right side of a contour is the inside and the left side of a contour is the outside (which provides the correct behavior for TrueType fonts). When the `gxAutoInsetStyle` flag is set, QuickDraw GX finds the true inside of each contour, regardless of the contour direction.

Setting both the `gxInsideFrameStyle` and `gxOutsideFrameStyle` style attributes results in the `inconsistent_parameters` error.

See “Grids” on page 3-20 and “Constraining Shape Geometries to Grids” beginning on page 3-40 for details about how QuickDraw GX constrains shapes to a grid. See “The Geometric Pen” on page 3-15 and “Manipulating Pen Width and Placement” on page 3-51 for examples of pen placement.

## The Cap Structure

---

QuickDraw GX allows you to specify what to draw at the start and at the end of a shape's contours. In particular, you may specify a start cap for any point shape, and you may specify a start cap and an end cap for any line, curve, polygon, or path shape that has an open-frame shape fill.

QuickDraw GX uses the cap property of a shape's style object to store information about the start cap and end cap of the shape.

You use the cap structure when specifying cap information (using the `GXSetStyleCap` or `GXSetShapeCap` functions) and when retrieving cap information (using the `GXGetStyleCap` or `GXGetShapeCap` functions).

## Geometric Styles

The cap structure is defined by the `gxCapRecord` data type:

```
struct gxCapRecord {
    gxCapAttribute attributes;
    gxShape          startCap;
    gxShape          endCap;
};
```

**Field descriptions**

<code>attributes</code>	Modifies the behavior of the caps. The next section, “Cap Attributes,” describes the <code>gxCapAttribute</code> flags in detail.
<code>startCap</code>	Specifies what the start cap should look like. You must use shapes in their primitive form for the start cap shape. (Primitive shapes are described in detail in Chapter 4, “Geometric Operations,” in this book.) You may not use framed shapes, shapes with an inverse shape fill, full shapes, text shapes, glyph shapes, layout shapes, bitmap shapes, or picture shapes as the start cap shape.  QuickDraw GX considers only the geometric properties (the shape type, the shape fill, and the shape geometry) of the shape specified by the <code>startCap</code> field. QuickDraw GX ignores the owner count, shape tags, and shape attributes properties and the style, ink, and transform objects of the start cap shape.
<code>endCap</code>	Specifies what the end cap should look like. You must use shapes in their primitive form for the end cap shape. (Primitive shapes are described in detail in Chapter 4, “Geometric Operations,” in this book.) You may not use framed shapes, shapes with an inverse shape fill, full shapes, text shapes, glyph shapes, layout shapes, bitmap shapes, or picture shapes as the end cap shape.  QuickDraw GX considers only the geometric properties (the shape type, the shape fill, and the shape geometry) of the shape specified by the <code>endCap</code> field. QuickDraw GX ignores the owner count, shape tags, and shape attributes properties and the style, ink, and transform objects of the end cap shape.

See “Caps” beginning on page 3-23, “Adding Caps to a Shape” beginning on page 3-57, and “Adding Standard Caps to a Shape” beginning on page 3-59 for examples of caps.

## Cap Attributes

---

Each cap structure contains a set of flags that modify the way a shape is capped. These constants are defined in the `gxCapAttributes` enumeration:

```
enum gxCapAttributes {
    gxLevelStartCap= 0x0001;
    gxLevelEndCap  = 0x0002;
};

typedef long gxCapAttribute;
```

### Constant descriptions

<code>gxLevelStartCap</code>	Suppresses rotation of the start cap shape. When you set this flag, QuickDraw GX does not rotate the start cap shape to match the slope of the capped contour. Instead, QuickDraw GX places the start cap shape onto the start of the capped contour with the exact orientation specified by the start cap shape's geometry.
<code>gxLevelEndCap</code>	Suppresses rotation of the end cap shape. When you set this flag, QuickDraw GX does not rotate the end cap shape to match the slope of the capped contour. Instead, QuickDraw GX places the end cap shape onto the start of the capped contour with the exact orientation specified by the end cap shape's geometry.

## The Join Structure

---

QuickDraw GX allows you to specify a join shape to be drawn at the corners of another shape's contours. In particular, you may specify a join shape for any rectangle, polygon, or path shape that has an open-frame shape fill or a closed-frame shape fill.

- n For shapes with the closed-frame shape fill, QuickDraw GX draws the specified join shape at every on-curve geometric point of each contour.
- n For shapes with the open-frame shape fill, QuickDraw GX draws the specified join shape at every on-curve geometric point between the first point and the last point of each contour.

QuickDraw GX uses the join property of a shape's style object to store information about the join of the shape.

You use the join structure when specifying join information (using the `GXSetStyleJoin` or `GXSetShapeJoin` functions) and when retrieving join information (using the `GXGetStyleJoin` or `GXGetShapeJoin` functions).

The join structure is defined by the `gxJoinRecord` data type:

```
struct gxJoinRecord {
    gxJoinAttribute attributes;
    gxShape          join;
    Fixed            miter;
};
```

#### Field descriptions

<code>attributes</code>	Allows you to specify a level join, or to specify one of two standard types of joins: sharp joins and curve joins. The next section, “Join Attributes,” describes the <code>gxJoinAttribute</code> flags in detail.
<code>join</code>	Specifies what the join should look like. You must use shapes in their primitive form for the join shape. (Primitive shapes are described in detail in Chapter 4, “Geometric Operations,” in this book.) You may not use framed shapes, shapes with an inverse shape fill, full shapes, text shapes, glyph shapes, layout shapes, bitmap shapes, or picture shapes as the join shape.  QuickDraw GX considers only the geometric properties (the shape type, the shape fill, and the shape geometry) of the shape specified by the <code>join</code> field. QuickDraw GX ignores the owner count, shape tags, and shape attributes properties and the style, ink, and transform objects of the join shape.  You set this field to <code>nil</code> when you want to specify a standard join: a sharp join or curve join.
<code>miter</code>	Used to truncate sharp joins. See the next section, “Join Attributes,” for more information about sharp joins.

See “Joins” beginning on page 3-25, “Adding Joins to a Shape” beginning on page 3-61, and “Adding Standard Joins to a Shape” beginning on page 3-64 for examples of joins.

## Join Attributes

---

Each join structure contains a set of flags that allow you to specify level joins, sharp joins, and curve joins. These constants are defined in the `gxJoinAttributes` enumeration:

```
enum gxJoinAttributes {
    gxSharpJoin= 0x0000,
    gxCurveJoin= 0x0001,
    gxLevelJoin= 0x0002
};

typedef long gxJoinAttribute;
```



**Constant descriptions**

<code>gxSharpJoin</code>	Indicates that QuickDraw GX should continue the outside edges of the corners of the joined shape until they meet at a point. You can use the <code>miter</code> field of the join structure to limit the size of a sharp join for very sharp corners.
<code>gxCurveJoin</code>	Indicates that QuickDraw GX should connect the outside edges of the corners of the joined shape with a circular curve.
<code>gxLevelJoin</code>	Suppresses rotation of the shape specified by the <code>join</code> field of the join structure. When you set this flag, QuickDraw GX does not rotate the join shape to match the mid-angle of the joined corner. Instead, QuickDraw GX places the join shape onto the joined corner with the exact orientation specified by the geometry of the join shape.

QuickDraw GX draws a sharp join or a curve join for every corner of every geometric shape; you may additionally specify a join shape to be added to a shape's corner using the `join` field of the join structure.

The `miter` field of the join structure allows you to limit the size of sharp joins, which is particularly useful if the joined corner is very sharp. In the `miter` field, you specify the maximum distance between the actual corner (as specified by the joined shape's geometry) and the tip of the sharp corner as drawn.

See “Adding Standard Joins to a Shape” beginning on page 3-64 for an example of a standard join.

## The Dash Structure

---

With QuickDraw GX, you can specify that certain shapes should be drawn with dashed, instead of solid, contours. In particular, you may specify a dash for any line, curve, rectangle, polygon, or path shape that has an open-frame shape fill or a closed-frame shape fill.

QuickDraw GX uses the dash property of a shape's style object to store information about how to dash the shape.

You use the dash structure when specifying dash information (using the `GXSetStyleDash` or `GXSetShapeDash` functions) and when retrieving dash information (using the `GXGetStyleDash` or `GXGetShapeDash` functions).

The dash structure is defined by the `gxDashRecord` data type:

```
struct gxDashRecord {
    gxDashAttribute    attributes;
    gxShape             dash;
    Fixed              advance;
    fract              phase;
    Fixed              scale;
};
```

## Geometric Styles

**Field descriptions**

attributes	Modifies the behavior of the dashes. The next section, “Dash Attributes,” describes the <code>gxDashAttribute</code> flags in detail.
dash	Specifies what the dash should look like. You must use shapes in their primitive form for the dash shape. (Primitive shapes are described in detail in Chapter 4, “Geometric Operations,” in this book.) You may not use text shapes, layout shapes, bitmap shapes, or picture shapes as the dash shape. However, you may use framed shapes and glyph, and you may also use shapes with an inverse shape fill if the clip dash attribute is set.  QuickDraw GX considers only the geometric properties (the shape type, the shape fill, and the shape geometry) of the shape specified by the <code>dash</code> field. QuickDraw GX ignores the owner count, shape tags, and shape attributes properties and the style, ink, and transform objects of the dash shape.
advance	Indicates the distance between dashes. This fixed-point value is the distance along the contours of the dashed shape between the beginning of a dash and the beginning of the following dash. The value must be greater than 0.
phase	Specifies the initial placement of a dash. This value can vary between -2.0 and 2.0. A value of 0 indicates that the dash shape should not be offset—that is, the start of the first dash shape should be aligned with the start of the contour. A value greater than 0 indicates that the first dash along the contour should begin a certain percentage into the dash shape. A value of 1.0 indicates that the dashes should be shifted exactly one advance width—this value is equivalent to specifying a value of 0. Values greater than 1.0 are equivalent to their fractional part.
scale	Specifies the scaling of the dash shape. QuickDraw GX scales the dash shape in one dimension—perpendicularly to the contour being dashed. The factor it uses to scales the dash shape in this dimension is the pen width divided by the dash scale. Therefore, decreasing the dash scale has the effect of thickening the dashed contour.

See “Dashes” beginning on page 3-27 for more information about dashes, and see page 3-66 through page 3-81 for examples of dashing.

## Dash Attributes

Each dash structure contains a set of flags that modify the way a shape is dashed. These constants are defined in the `gxDashAttributes` enumeration:

```
enum gxDashAttributes {
    gxBendDash      = 0x0001;
    gxBreakDash     = 0x0002;
    gxClipDash      = 0x0004;
    gxLevelDash     = 0x0008;
    gxAutoAdvanceDash = 0x0010;
};
```

```
typedef long gxDashAttribute;
```

### Constant descriptions

<code>gxBendDash</code>	Distorts the dash shape to match the contour being dashed. A dash may have the <code>gxBendDash</code> attribute only when the dashed shape's pen width is zero, indicating hairline contours. (Any other pen width results in an error condition.) When the <code>gxBendDash</code> attribute is set, QuickDraw GX maps the dash shape onto the x-axis (so that it becomes one-dimensional) and bends this flattened dash shape along the contours of the shape being dashed.
<code>gxBreakDash</code>	Indicates that QuickDraw GX should rotate and place each contour of the dash shape separately. When this attribute is set, QuickDraw GX calculates the center point of each contour of the dash shape and rotates and centers it appropriately along the contour of the shape being dashed. See Figure 3-25 on page 3-30 for an example.
<code>gxClipDash</code>	Indicates that QuickDraw GX should clip the dashes to the pen width of the dashed shape. See Figure 3-24 on page 3-29 for an example. This attribute causes dashes to have some complicated interactions with caps and joins. See the section "Interactions Between Caps, Joins, Dashes, and Patterns" on page 3-33 and "Combining Caps, Joins, Dashes, and Patterns" beginning on page 3-91 for more information.
<code>gxLevelDash</code>	Suppresses rotation of the dash shape. When this attribute is set, QuickDraw GX does not rotate the dash shape to match the slope of the dashed shape's contours. Instead, QuickDraw GX places the dash shape onto the contours of the dashed shape with the exact orientation specified by the geometry of the dash shape.
<code>gxAutoAdvanceDash</code>	Adjusts the dash advance so that a whole multiple of dash shapes fit each contour.

These sections include examples of using dash attributes:

- n “Dashing a Shape” on page 3-66
- n “Adjusting Dashes to Fit Contours” on page 3-70
- n “Breaking and Bending Dashes” on page 3-74

## The Pattern Structure

---

With QuickDraw GX, you can specify that certain shapes be patterned. For shapes with solid shape fills, QuickDraw GX fills the shape by repeating a pattern shape that you specify, over a grid that you specify.

You can also pattern shapes with framed shape fills. For example, imagine a rectangle shape with the closed-frame shape fill and a pen width of 20. If you patterned this rectangle, QuickDraw GX would fill the frame of the rectangle with the pattern. See the section “Adding a Pattern to a Shape” on page 3-86 for examples.

QuickDraw GX uses the pattern property of a shape’s style object to store information about how to pattern the shape.

You use the pattern structure when specifying pattern information (using the `GXSetStylePattern` or `GXSetShapePattern` functions) and when retrieving pattern information (using the `GXGetStylePattern` or `GXGetShapePattern` functions).

The pattern structure is defined by the `gxPatternRecord` data type:

```
struct gxPatternRecord {
    gxPatternAttribute attributes;
    gxShape              pattern;
    gxPoint              u;
    gxPoint              v;
};
```

### Field descriptions

attributes	Modifies the behavior of the pattern. The next section, “Pattern Attributes,” describes the <code>gxPatternAttribute</code> flags in detail.
pattern	Specifies the shape that makes up the pattern. You must use shapes in their primitive form for the pattern shape. (Primitive shapes are described in detail in Chapter 4, “Geometric Operations,” in this book.) You may not use text shapes, layout shapes, or picture shapes as the pattern shape. However, you may use framed shape shapes and shapes with an inverse shape fill. You may also use bitmap shapes with any pixel size as long as the bitmap shape does not contain color profile information.  QuickDraw GX considers only the geometric properties (the shape type, the shape fill, and the shape geometry) of the shape specified by the pattern field. QuickDraw GX ignores the owner count, shape tags, and shape attributes properties and the style, ink, and transform objects of the pattern shape.

<code>u</code>	One of a pair of vectors that determine how QuickDraw GX places the pattern shape. This field, along with the <code>v</code> field, defines the pattern grid.
<code>v</code>	The other of the pair of vectors that describe how QuickDraw GX places the pattern shape. This field, along with the <code>u</code> field, defines the pattern grid.

The `u` and `v` fields together form a pair of vectors that define the pattern grid, which determines where QuickDraw GX places the pattern shape. The vectors define a grid of parallelograms and QuickDraw GX draws a pattern shape at every intersection in this grid.

The vectors specified by the `u` and `v` fields do not need to be any order, but they must point in different directions—that is, they may not lie on the same line. If you specify `u` and `v` vectors that are parallel, a `pattern_lattice_out_of_range` error results.

#### Optimization Note

QuickDraw GX draws bitmap patterns very quickly—that is, nearly as fast as a nonpatterned fill—if the `u` and `v` vectors place the patterns in a rectangular grid the size of the bitmap. `u`

See “Patterns” beginning on page 3-31 for more information about patterns and the pattern grid, and “Adding a Pattern to a Shape” on page 3-86 for an example of using patterns.

## Pattern Attributes

---

Each pattern structure contains a set of flags that modify the way a shape is patterned. These constants are defined in the `gxPatternAttributes` enumeration:

```
enum gxPatternAttributes {
    gxPortAlignPattern = 0x0001,
    gxPortMapPattern   = 0x0002
};

typedef long gxPatternAttribute;
```

#### Constant descriptions

`gxPortAlignPattern`

Indicates that QuickDraw GX should align the pattern shapes with the view device instead of the patterned shape. When this attribute is set, the pattern does not move when the patterned shape moves. Instead, the position of the pattern stays constant with respect to the view device. In effect, the patterned shape allows you to see through to a constant background covered by the pattern shape.

`gxPortMapPattern`

Indicates that mappings in the patterned shape's transform affect the patterned shape but do not affect the pattern. As an example, imagine that the transform of the patterned shape specifies that the patterned shape be scaled up by a factor of 2. If the `gxPortMapPattern` attribute is not set, then the pattern itself is magnified as well as the patterned shape. If this attribute is set, then the pattern stays the same size, but the patterned shape shows more of the pattern.

See the section "Patterns" on page 3-31 for an example of these attributes.

## Functions

---

This section describes the functions available for manipulating a style object's geometric properties:

- n the style attributes
- n the curve error
- n the pen width
- n the caps
- n the join
- n the dash
- n the pattern

These properties together determine the stylistic variations applied to the frame and the area of a shape when drawn.

For information about creating, disposing of, copying, and comparing style objects as well as information about manipulating style tags and style owner counts, see *Inside Macintosh: QuickDraw GX Objects*.

For information about the typographic style properties, such as font, text size, and text face, see *Inside Macintosh: QuickDraw GX Typography*.

In general, there are two types of functions that manipulate the properties of style objects:

- n functions that require you to provide a reference to the style object itself
- n functions that allow you to provide a reference to a shape and affect the style object associated with that shape

The section "Associating Styles With Shapes" on page 3-36 provides an example of both of these types of functions and compares their results.

Both types of functions are described in this reference section.

## Getting and Setting Style Attributes

---

The style attributes are a set of flags that modify the behavior of the style object. In particular, these flags allow you to specify how QuickDraw GX places the geometric pen with respect to a shape's geometry and whether the shape should be constrained to a grid.

For a description of the style attributes, see the section “Style Attributes” on page 3-98.

You can use the `GXGetStyleAttributes` function to find the style attributes of an existing style and the `GXSetStyleAttributes` function to set the style attributes of a style.

The `GXGetShapeStyleAttributes` and `GXSetShapeStyleAttributes` functions provide a way to determine and change the style attributes of a style object associated with a particular shape.

## GXGetStyleAttributes

---

You can use the `GXGetStyleAttributes` function to determine which style attributes are set for a particular style object.

```
gxStyleAttribute GXGetStyleAttributes(gxStyle source);
```

**source**            A reference to the style object whose style attributes you want to determine.

*function result*   The style attributes associated with the source style object.

### DESCRIPTION

The `GXGetStyleAttributes` function returns as its function result the style attributes of the style object specified by the `source` parameter.

As an example, to examine the `gxSourceGridStyle` flag of a style object referenced by the variable `source`, you could use this code:

```
if (GXGetStyleAttributes(source) & gxSourceGridStyle) {
    /* style has gxSourceGridStyle attribute set */
}
```

## Geometric Styles

The `gxCenterFrameStyle` attribute is set only if both the `gxInsideFrameStyle` and the `gxOutsideFrameStyle` attributes are clear, so if you want to test for a centered frame style you need this code:

```
if (GXGetStyleAttributes(source) &
    (gxInsideFrameStyle | gxOutsideFrameStyle) ==
    gxCenterFrameStyle) {
    /* style has gxCenterFrameStyle attribute set */
}
```

## ERRORS, WARNINGS, AND NOTICES

**Errors**

`out_of_memory`  
`style_is_nil`

## SEE ALSO

For a discussion of style attributes, see “Style Attributes” on page 3-98.

For an example of pen placement, see “Manipulating Pen Width and Placement” on page 3-51.

For an example of constraining shapes to grids, see “Constraining Shape Geometries to Grids” on page 3-40 and “Constraining Shapes to Device Grids” on page 3-42.

To examine the style attributes of a style object associated with a particular shape, use the `GXGetShapeStyleAttributes` function, which is described on page 3-112.

To alter the style attributes of a style object, use the `GXSetStyleAttributes` function, which is described in the next section.

To alter the style attributes of a style object associated with a particular shape, use the `GXSetShapeStyleAttributes` function, which is described on page 3-113.

## GXSetStyleAttributes

---

You can use the `GXSetStyleAttributes` function to alter the style attributes for a particular style object.

```
void GXSetStyleAttributes(gxStyle target,
                          gxStyleAttribute attributes);
```

`target`            A reference to the style object whose attributes you want to alter.

`attributes`        The new set of attributes.



**DESCRIPTION**

The `GXSetStyleAttributes` function sets the style attributes of the style object specified by the `target` parameter to be the attributes specified in the `attributes` parameter.

You can use this function in combination with the `GXGetStyleAttributes` function to set or clear single style attributes. For example, to clear the `gxSourceGridStyle` attribute of a style object referenced by the variable `target`, you could use this line of code:

```
GXSetStyleAttributes(target,
                    GXGetStyleAttributes(target & ~gxSourceGridStyle);
```

To set the `gxSourceGridStyle` attribute, you could use this line of code:

```
GXSetStyleAttributes(target,
                    GXGetStyleAttributes(target | gxSourceGridStyle);
```

To set the `gxCenterFrameStyle` attribute, you need to clear the `gxInsideFrameStyle` and `gxOutsideFrameStyle` attributes.

When you set a style's attributes using this function, you are effectively changing the style attributes for all shapes that share the style.

**ERRORS, WARNINGS, AND NOTICES****Errors**

<code>out_of_memory</code>	
<code>style_is_nil</code>	
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)

**Notices (debugging version)**

<code>attributes_already_set</code>
-------------------------------------

**SEE ALSO**

For a discussion of style attributes, see “Style Attributes” on page 3-98.

For an example of pen placement, see “Manipulating Pen Width and Placement” beginning on page 3-51.

For an example of constraining shapes to grids, see “Constraining Shape Geometries to Grids” on page 3-40 and “Constraining Shapes to Device Grids” on page 3-42.

To examine the style attributes of a style object, use the `GXGetStyleAttributes` function, which is described on page 3-109.

To examine the style attributes of a style object associated with a particular shape, use the `GXGetShapeStyleAttributes` function, which is described in the next section. To alter the style attributes of a style object associated with a particular shape, use the `GXSetShapeStyleAttributes` function, which is described on page 3-113.

## GXGetShapeStyleAttributes

---

You can use the `GXGetShapeStyleAttributes` function to determine which style attributes are set for the style object of a particular shape.

```
gxStyleAttribute GXGetShapeStyleAttributes(gxShape source);
```

**source**      A reference to the shape whose style attributes you want to determine.

*function result*   The style attributes of the source shape's style object.

### DESCRIPTION

The `GXGetShapeStyleAttributes` function provides a convenient way to determine the style attributes of a shape without having to call the `GXGetShapeStyle` function to obtain a reference to the shape's style object.

As an example, to examine the `gxSourceGridStyle` flag of a style object associated with the shape object referenced by the variable `source`, you could use this code:

```
if (GXGetShapeStyleAttributes(source) & gxSourceGridStyle) {
    /* shape's style has gxSourceGridStyle attribute set */
}
```

The `gxCenterFrameStyle` attribute is set only if both the `gxInsideFrameStyle` and the `gxOutsideFrameStyle` attributes are clear, so if you want to test for a centered frame style you need this code:

```
if (GXGetShapeStyleAttributes(source) &
    (gxInsideFrameStyle | gxOutsideFrameStyle) ==
    gxCenterFrameStyle) {
    /* shape's style has gxCenterFrameStyle attribute set */
}
```

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`  
`shape_is_nil`

### SEE ALSO

For a discussion of style attributes, see “Style Attributes” on page 3-98.

For an example of pen placement, see “Manipulating Pen Width and Placement” beginning on page 3-51.

For an example of constraining shapes to grids, see “Constraining Shape Geometries to Grids” on page 3-40 and “Constraining Shapes to Device Grids” on page 3-42.

To examine or alter the style attributes of a style object, use the `GXGetStyleAttributes` function, which is described on page 3-109. To alter the style attributes of a style object, use the `GXSetStyleAttributes` function, which is described on page 3-110.

To alter the style attributes of a style object associated with a particular shape, use the `GXSetShapeStyleAttributes` function, which is described in the next section.

## **GXSetShapeStyleAttributes**

---

You can use the `GXSetShapeStyleAttributes` function to alter the style attributes of the style object associated with a particular shape.

```
void GXSetShapeStyleAttributes(gxShape target,
                               gxStyleAttribute attributes);
```

`target`            A reference to the shape whose style attributes you want to alter.

`attributes`        The new set of attributes.

### **DESCRIPTION**

The `GXSetShapeStyleAttributes` function sets the style attributes of the style object associated with the shape specified by the `target` parameter.

If the `target` shape shares its style object with other shapes, this function makes a copy of the style object, sets the `target` shape to reference the copy, and changes the style attributes of the copy. (However, if the effect of this function would leave the style attributes unchanged, this function does not create a copy of the style object; instead, it posts a notice).

You can use this function in combination with the `GXGetShapeStyleAttributes` function to set or clear single style attributes. For example, to clear the `gxSourceGridStyle` attribute of a style object associated with a `target` shape, you could use this line of code:

```
GXSetShapeStyleAttributes(target,
                           GXGetShapeStyleAttributes(target & ~gxSourceGridStyle);
```

To set the `gxSourceGridStyle` attribute, you could use this line of code:

```
GXSetShapeStyleAttributes(target,
                           GXGetShapeStyleAttributes(target | gxSourceGridStyle);
```

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`  
`style_is_nil`  
`inconsistent_parameters` (debugging version)  
`parameter_out_of_range` (debugging version)

**Notices (debugging version)**

`attributes_already_set`

**SEE ALSO**

For a discussion of style attributes, see “Style Attributes” on page 3-98.

For an example of pen placement, see “Manipulating Pen Width and Placement” on page 3-51.

For an example of constraining shapes to grids, see “Constraining Shape Geometries to Grids” on page 3-40 and “Constraining Shapes to Device Grids” on page 3-42.

To examine the style attributes of a style object associated with a particular shape, use the `GXGetShapeStyleAttributes` function, which is described on page 3-112.

To examine the style attributes of a style object, use the `GXGetStyleAttributes` function, which is described on page 3-109. To alter the style attributes of a style object, use the `GXSetStyleAttributes` function, which is described on page 3-110.

**Getting and Setting Curve Error**

---

The curve error property of style objects specifies the allowable amount of error when QuickDraw GX converts a path shape into a polygon shape. It also specifies how far apart geometric points must be for QuickDraw GX to consider them separate points when performing geometric operations on shapes or reducing shapes.

For example, when you call the `GXInsetShape` function on a tight curve, the resulting curve can require many more geometric points than the original curve. QuickDraw GX simplifies the resulting shape by removing geometric points that are within the shape’s curve error from another geometric point.

You can use the `GXGetStyleCurveError` function to determine the curve error of a style object and the `GXSetStyleCurveError` function to change the curve error of a style object.

The `GXGetShapeCurveError` and `GXSetShapeCurveError` functions provide a way to determine and change the curve error of the style object associated with a particular shape.

## GXGetStyleCurveError

---

You can use the `GXGetStyleCurveError` function to determine the curve error of a style object.

```
Fixed GXGetStyleCurveError(gxStyle source);
```

**source**        A reference to the style object whose curve error you want to determine.

*function result*   The curve error of the source style object.

### DESCRIPTION

When a path shape has a curve error of 0, QuickDraw GX does not approximate the path shape with a polygon shape when converting it to a polygon. Instead, QuickDraw GX simply removes off-curve control points, as shown in “Using Curve Error When Converting Paths to Polygons” on page 3-45.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`  
`style_is_nil`

### SEE ALSO

For a discussion of curve error, see “Curve Error” on page 3-14.

For examples of using curve error, see “Using Curve Error When Converting Paths to Polygons” on page 3-45 and “Using Curve Error When Reducing Shapes” on page 3-49.

To change the curve error of a style object, use the `GXSetStyleCurveError` function, which is described in the next section.

To examine the curve error of a style object associated with a particular shape, use the `GXGetShapeCurveError` function, which is described on page 3-117. To change the curve error of a style object associated with a particular shape, use the `GXSetShapeCurveError` function, which is described on page 3-118.

## GXSetStyleCurveError

---

You can use the `GXSetStyleCurveError` function to change the curve error of a style object.

```
void GXSetStyleCurveError(gxStyle target, Fixed error);
```

`target`        A reference to the style object whose curve error you want to change.  
`error`         The new curve error.

### DESCRIPTION

This routine sets the curve error of the style object specified by the `target` parameter to be the fixed-point value specified by the `error` parameter. You may specify any nonnegative value for this parameter.

When a path shape has a curve error of 0.0, QuickDraw GX does not approximate the path shape with a polygon shape when converting it to a polygon. Instead, QuickDraw GX simply removes off-curve control points, as shown in “Using Curve Error When Converting Paths to Polygons” on page 3-45.

A very small curve error may cause the `GXSetShapeType` function and certain geometric operations such as the `GXInsetShape` function to use inappropriate amounts of memory and time.

When you set a style’s curve error using this function, you are effectively changing the curve error for all shapes that share the style.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`  
`style_is_nil`  
`parameter_out_of_range`        (debugging version)

#### Notices (debugging version)

`curve_error_already_set`

### SEE ALSO

For a discussion of curve error, see “Curve Error” on page 3-14.

For examples of curve error, see “Using Curve Error When Converting Paths to Polygons” on page 3-45 and “Using Curve Error When Reducing Shapes” on page 3-49.

To determine the curve error of a style object, use the `GXGetStyleCurveError` function, which is described on page 3-115.

To examine the curve error of a style object associated with a particular shape, use the `GXGetShapeCurveError` function, which is described in the next section. To change the curve error of a style object associated with a particular shape, use the `GXSetShapeCurveError` function, which is described on page 3-118.

## GXGetShapeCurveError

---

You can use the `GXGetShapeCurveError` function to determine the curve error of the style object associated with a particular shape.

```
Fixed GXGetShapeCurveError(gxShape source);
```

**source**        A reference to the shape whose curve error you want to determine.

**function result**   The curve error of the style object associated with the source shape.

### DESCRIPTION

When a path shape has a curve error of 0, QuickDraw GX does not approximate the path shape with a polygon shape when converting it to a polygon. Instead, QuickDraw GX simply removes off-curve control points, as shown in “Using Curve Error When Converting Paths to Polygons” on page 3-45.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`  
`shape_is_nil`

### SEE ALSO

For a discussion of curve error, see “Curve Error” on page 3-14.

For examples of curve error, see “Using Curve Error When Converting Paths to Polygons” on page 3-45 and “Using Curve Error When Reducing Shapes” on page 3-49.

To determine the curve error of a style object, use the `GXGetStyleCurveError` function, which is described on page 3-115. To change the curve error of a style object, use the `GXSetStyleCurveError` function, which is described on page 3-116.

To change the curve error of a style object associated with a particular shape, use the `GXSetShapeCurveError` function, which is described in the next section.

## GXSetShapeCurveError

---

You can use the `GXSetShapeCurveError` function to change the curve error of the style object associated with a particular shape.

```
void GXSetShapeCurveError(gxShape target, Fixed error);
```

**target**            A reference to the shape whose curve error you want to change.  
**error**            The new curve error.

### DESCRIPTION

The `GXSetShapeCurveError` function replaces the curve error of the style object associated with the shape specified by the `source` parameter with the value in the `error` parameter. You may specify any nonnegative value for this parameter.

If the target shape shares its style object with other shapes, this function makes a copy of the style object, sets the target shape to reference the copy, and changes the curve error of the copy. (However, if the effect of this function would leave the curve error unchanged, this function does not create a copy of the style object; instead, it posts a notice.)

When the curve error is 0, QuickDraw GX does not approximate a path shape with a polygon shape when converting from a path to a polygon. Instead, QuickDraw GX simply removes off-curve control points, as shown in “Using Curve Error When Converting Paths to Polygons” on page 3-45.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`  
`shape_is_nil`  
`parameter_out_of_range`            (debugging version)

#### Notices (debugging version)

`curve_error_already_set`

### SEE ALSO

For a discussion of curve error, see “Curve Error” on page 3-14.

For examples of curve error, see “Using Curve Error When Converting Paths to Polygons” on page 3-45 and “Using Curve Error When Reducing Shapes” on page 3-49.

To determine the curve error of a style object, use the `GXGetStyleCurveError` function, which is described on page 3-115. To change the curve error of a style object, use the `GXSetStyleCurveError` function, which is described on page 3-116.

To determine the curve error of a style object associated with a particular shape, use the `GXGetShapeCurveError` function, which is described on page 3-117.



## Getting and Setting the Pen Width

---

The pen width property of a style object specifies the width at which QuickDraw GX should draw a shape's contours. A pen width of 0 specifies a hairline. QuickDraw GX always draws hairlines at the resolution of the output device—one pixel wide. The pen width also affects dashing: QuickDraw GX scales a shape's dashes (in the y-coordinate direction) by the pen width. Also, the pen width affects the clip shape that QuickDraw GX uses to clip the dashes when a shape's clip dash attribute is set.

You can use the `GXGetStylePen` function to determine the pen width of a style object and the `GXSetStylePen` function to change the pen width of a style object.

The `GXGetShapePen` and `GXSetShapePen` functions provide a way to determine and change the pen width of the style object associated with a particular shape.

## GXGetStylePen

---

You can use the `GXGetStylePen` function to determine the pen width of a particular style object.

```
Fixed GXGetStylePen(gxStyle source);
```

*source*      A reference to the style object whose pen width you want to determine.

*function result*   The pen width of the source style object.

### DESCRIPTION

A pen width of 0.0 indicates a hairline width; QuickDraw GX always draws hairlines one pixel wide.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`  
`style_is_nil`

### SEE ALSO

For a discussion of the drawing pen, see “The Geometric Pen” on page 3-15.

For an example of changing a shape's pen width, see “Manipulating Pen Width and Placement” on page 3-51.

To change the pen width of a style object, use the `GXSetStylePen` function, which is described in the next section.

To determine the pen width of a style object associated with a particular shape, use the `GXGetShapePen` function, which is described on page 3-121. To change the pen width of a style object associated with a particular shape, use the `GXSetShapePen` function, which is described on page 3-122.

## GXSetStylePen

---

You can use the `GXSetStylePen` function to change the pen width of a style object.

```
void GXSetStylePen(gxStyle target, Fixed pen);
```

<code>target</code>	A reference to the style object whose pen width you want to change.
<code>pen</code>	The new pen width.

### DESCRIPTION

The `GXSetStylePen` function sets the pen width of the style object specified by the `target` parameter to the value specified in the `pen` parameter. You may specify any nonnegative value for this parameter.

A pen width of 0 indicates a hairline; QuickDraw GX always draws hairlines one pixel wide.

Remember that the `pen` parameter is specified as a fixed-point value. Very small diameters may cause all drawing to disappear, since a shape may fall between pixels. A common mistake when setting the pen width is to specify the pen width as an integer, rather than a fixed-point value:

```
GXSetStylePen(myStyle, 1); /* set the pen width to 1/65536 */
GXSetStylePen(myStyle, ff(1)); /* set the pen width to 1.0 */
```

When you set the pen width using this function, you are effectively changing the pen width for all shapes that share the style.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

<code>out_of_memory</code>	
<code>style_is_nil</code>	
<code>parameter_out_of_range</code>	(debugging version)

#### Notices (debugging version)

<code>pen_size_already_set</code>
-----------------------------------

**SEE ALSO**

For a discussion of the drawing pen, see “The Geometric Pen” on page 3-15.

For an example of changing a shape’s pen width, see “Manipulating Pen Width and Placement” on page 3-51.

To determine the pen width of a style object, use the `GXGetStylePen` function, which is described on page 3-119.

To determine the pen width of a style object associated with a particular shape, use the `GXGetShapePen` function, which is described in the next section. To change the pen width of a style object associated with a particular shape, use the `GXSetShapePen` function, which is described on page 3-122.

**GXGetShapePen**

---

You can use the `GXGetShapePen` function to determine the pen width of the style object associated with a particular shape.

```
Fixed GXGetShapePen(gxShape source);
```

*source*      A reference to the shape whose pen width you want to determine.

*function result*   The pen width of the source shape’s style object.

**DESCRIPTION**

A pen width of 0.0 indicates a hairline width; QuickDraw GX always draws hairlines one pixel wide.

**ERRORS, WARNINGS, AND NOTICES****Errors**

```
out_of_memory
shape_is_nil
```

**SEE ALSO**

For a discussion of the drawing pen, see “The Geometric Pen” on page 3-15.

For an example of changing a shape’s pen width, see “Manipulating Pen Width and Placement” on page 3-51.

To determine the pen width of a style object, use the `GXGetStylePen` function, which is described on page 3-119. To change the pen width of a style object, use the `GXSetStylePen` function, which is described on page 3-120.

To change the pen width of a style object associated with a particular shape, use the `GXSetShapePen` function, which is described in the next section.

## GXSetShapePen

---

You can use the `GXSetShapePen` function to change the pen width of the style object associated with a particular shape.

```
void GXSetShapePen(gxShape target, Fixed pen);
```

`target`      A reference to the shape whose pen width you want to change.

`pen`          The new pen width.

### DESCRIPTION

The `GXSetShapePen` function sets the pen width of the target shape's style object to be the value specified in the `pen` parameter. You may specify any nonnegative value for this parameter.

If the target shape shares its style object with other shapes, this function makes a copy of the style object, sets the target shape to reference the copy, and changes the pen width of the copy. (However, if the effect of this function would leave the pen width information unchanged, this function does not create a copy of the style object; instead, it posts a notice.)

A pen width of 0 indicates a hairline; QuickDraw GX always draws hairlines one pixel wide.

```
GXSetShapePen(myShape, 0); /* set as thin as renderable */
```

Remember that the `pen` parameter is specified as a fixed-point value. Very small diameters may cause all drawing to disappear, since a shape may fall between pixels. A common mistake when setting the pen width is to specify the pen width as an integer, rather than a fixed-point value:

```
GXSetStylePen(myStyle, 1); /* set the pen width to 1/65536 */
```

```
GXSetStylePen(myStyle, ff(1)); /* set the pen width to 1.0 */
```

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`  
`shape_is_nil`  
`parameter_out_of_range`      (debugging version)

**Notices (debugging version)**

`pen_size_already_set`

**SEE ALSO**

For a discussion of the drawing pen, see “The Geometric Pen” on page 3-15.

For an example of changing a shape’s pen width, see “Manipulating Pen Width and Placement” on page 3-51.

To determine the pen width of a style object, use the `GXGetStylePen` function, which is described on page 3-119. To change the pen width of a style object, use the `GXSetStylePen` function, which is described on page 3-120.

To determine the pen width of a style object associated with a particular shape, use the `GXGetShapePen` function, which is described on page 3-121.

**Getting and Setting Caps**

---

QuickDraw GX allows you to specify what to draw at the start and at the end of a shape’s contours. In particular, you may specify a start cap for any point shape, and you may specify a start cap and an end cap for any line, curve, polygon, or path shape that has an `gxOpenFrameFill` shape fill. You must always specify cap shapes in primitive form.

“The Cap Structure” on page 3-99 describes the `gxCapRecord` structure, which you use when retrieving or specifying cap information. That section also describes what types of shapes you may use as cap shapes.

You can use the `GXGetStyleCap` function to retrieve the cap information from a style object and the `GXSetStyleCap` function to specify cap information for a style object.

The `GXGetShapeCap` and `GXSetShapeCap` functions provide a way to retrieve and specify cap information for the style object associated with a particular shape.

## GXGetStyleCap

---

You can use the `GXGetStyleCap` function to retrieve the cap information from a style object.

```
gxCapRecord *GXGetStyleCap(gxStyle source, gxCapRecord *cap);
```

`source`      The style object whose cap information you want to retrieve.

`cap`          A pointer to a `gxCapRecord` structure. On return, this structure contains the cap information for the source style object.

*function result*   A copy of the `gxCapRecord` associated with the source style.

### DESCRIPTION

The `GXGetStyleCap` function returns as its function result, and in the `cap` parameter, a `gxCapRecord` structure containing the cap information for the style object specified by the `source` parameter.

This function creates new shapes to encapsulate the start cap and end cap geometries, and places references to these shapes in the `startCap` and `endCap` fields of the returned `gxCapRecord` structure. You should dispose of these shapes when you no longer need them.

Since this function copies the cap information from the source style object, you may make changes to the `gxCapRecord` structure returned by this function without affecting the source style's cap information. If you want to change the cap information in the source style, you must use the `GXSetStyleCap` function.

### SPECIAL CONSIDERATIONS

If no error results, the `GXGetStyleCap` function creates shapes; you are responsible for disposing of these shapes when you no longer need them. See *Inside Macintosh: QuickDraw GX Objects* for information about disposing QuickDraw GX objects.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`  
`style_is_nil`  
`parameter_is_nil`      (debugging version)

**SEE ALSO**

For a discussion of start and end caps, see “Caps” on page 3-23.

For examples of adding caps to a shape, see “Adding Caps to a Shape” on page 3-57 and “Adding Standard Caps to a Shape” on page 3-59.

For a discussion of the `gxCapRecord` structure and a description of what types of shapes you can use as cap shapes, see “The Cap Structure” on page 3-99.

To specify cap information for a style object, use the `GXSetStyleCap` function, which is described in the next section.

To retrieve cap information from a style object associated with a particular shape, use the `GXGetShapeCap` function, which is described on page 3-126. To specify cap information for a style object associated with a particular shape, use the `GXSetShapeCap` function, which is described on page 3-128.

## **GXSetStyleCap**

---

You can use the `GXSetStyleCap` function to change the cap information of a style object.

```
void GXSetStyleCap(gxStyle target, const gxCapRecord *cap);
```

`target`            The style object whose cap information you want to change.

`cap`                A pointer to the new cap information.

**DESCRIPTION**

The `GXSetStyleCap` function replaces the cap information in the style object specified by the `target` parameter with the cap information specified in the `cap` parameter. You use the `gxCapRecord` structure to provide cap information.

Passing `nil` for the `cap` parameter indicates that you want no caps and QuickDraw GX removes any cap information from the target style.

When you set a style’s cap property using this function, you are effectively changing the caps for all shapes that share the style.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>style_is_nil</code>	
<code>parameter_out_of_range</code>	(debugging version)
<code>empty_shape_not_allowed</code>	(debugging version)
<code>ignorePlatformShape_not_allowed</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>nil_style_in_glyph_not_allowed</code>	(debugging version)
<code>complex_glyph_style_not_allowed</code>	(debugging version)
<code>shapeFill_not_allowed</code>	(debugging version)

**Notices (debugging version)**

<code>caps_already_set</code>
<code>tags_in_shape_ignored</code>

## SEE ALSO

For a discussion of start and end caps, see “Caps” on page 3-23.

For examples of adding caps to a shape, see “Adding Caps to a Shape” on page 3-57 and “Adding Standard Caps to a Shape” on page 3-59.

For a discussion of the `gxCapRecord` structure and a description of what types of shapes you can use as cap shapes, see “The Cap Structure” on page 3-99.

To retrieve cap information from a style object, use the `GXGetStyleCap` function, which is described on page 3-124.

To retrieve cap information from a style object associated with a particular shape, use the `GXGetShapeCap` function, which is described in the next section. To specify cap information for a style object associated with a particular shape, use the `GXSetShapeCap` function, which is described on page 3-128.

**GXGetShapeCap**

---

You can use the `GXGetShapeCap` function to retrieve cap information from the style object of a particular shape.

```
gxCapRecord *GXGetShapeCap(gxShape source, gxCapRecord *cap);
```

**source**      A reference to the shape whose cap information you want to retrieve.

**cap**          A pointer to a `gxCapRecord` structure. On return, this structure contains the cap information for the source shape.

**function result**    A copy of the `gxCapRecord` structure associated with the source shape’s style object.



**DESCRIPTION**

The `GXGetShapeCap` function returns as its function result, and in the `cap` parameter, a `gxCapRecord` structure containing the cap information for the style object associated with the shape specified by the source parameter.

This function creates new shapes to encapsulate the start cap and end cap geometries, and places references to these shapes in the `startCap` and `endCap` fields of the returned `gxCapRecord` structure. You should dispose of these shapes when you no longer need them.

Since this function copies the cap information from the source shape's style, you may make changes to the `gxCapRecord` structure returned by this function without affecting the source shape's caps. If you want to change the cap information for the source shape, you must use the `GXSetShapeCap` function.

**SPECIAL CONSIDERATIONS**

Unless an error results, the `GXGetShapeCap` function creates shapes; you are responsible for disposing of these shapes when you no longer need them. See *Inside Macintosh: QuickDraw GX Objects* for information about disposing of QuickDraw GX objects.

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`  
`shape_is_nil`  
`parameter_is_nil`

**SEE ALSO**

For a discussion of start and end caps, see “Caps” on page 3-23.

For examples of adding caps to a shape, see “Adding Caps to a Shape” on page 3-57 and “Adding Standard Caps to a Shape” on page 3-59.

For a discussion of the `gxCapRecord` structure and a description of what types of shapes you can use as cap shapes, see “The Cap Structure” on page 3-99.

To retrieve cap information from a style object, use the `GXGetStyleCap` function, which is described on page 3-124. To specify cap information for a style object, use the `GXSetStyleCap` function, which is described on page 3-125.

To specify cap information for a style object associated with a particular shape, use the `GXSetShapeCap` function, which is described in the next section.

## GXSetShapeCap

---

You can use the `GXSetShapeCap` function to change the cap information of the style object associated with a particular shape.

```
void GXSetShapeCap(gxShape target, const gxCapRecord *cap);
```

`target`      A reference to the shape whose cap information you want to change.

`cap`          A pointer to the new cap information.

### DESCRIPTION

The `GXSetShapeCap` function replaces the cap information in the style object of the shape specified by the `target` parameter with the cap information specified in the `cap` parameter. You use the `gxCapRecord` structure to provide cap information.

Passing `nil` for the `cap` parameter indicates that you want no caps and QuickDraw GX removes any cap information from the target shape.

If the target shape shares its style object with other shapes, this function makes a copy of the style object, sets the target shape to reference the copy, and changes the cap property of the copy. (However, if the effect of this function would leave the cap information unchanged, this function does not create a copy of the style object; instead, it posts a notice.)

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`

`style_is_nil`

`parameter_out_of_range`

(debugging version)

`empty_shape_not_allowed`

(debugging version)

`ignorePlatformShape_not_allowed`

(debugging version)

`illegal_type_for_shape`

(debugging version)

`nil_style_in_glyph_not_allowed`

(debugging version)

`complex_glyph_style_not_allowed`

(debugging version)

`shapeFill_not_allowed`

(debugging version)

#### Notices (debugging version)

`caps_already_set`

`tags_in_shape_ignored`

### SEE ALSO

For a discussion of start and end caps, see “Caps” on page 3-23.

For examples of adding caps to shapes, see “Adding Caps to a Shape” on page 3-57 and “Adding Standard Caps to a Shape” on page 3-59.

For a discussion of the `gxCapRecord` structure and a description of what types of shapes you can use as cap shapes, see “The Cap Structure” on page 3-99.

To retrieve cap information from a style object, use the `GXGetStyleCap` function, which is described on page 3-124.

To specify cap information for a style object, use the `GXSetStyleCap` function, which is described on page 3-125.

To retrieve cap information from a style object associated with a particular shape, use the `GXGetShapeCap` function, which is described on page 3-126.

## Getting and Setting Joins

---

QuickDraw GX allows you to specify what to draw at corners of a shape's contours. In particular, you may specify a corner join for any rectangle, polygon, or path shape that has an open-frame shape fill or a closed-frame shape fill. You must always specify join shapes in primitive form.

"The Join Structure" on page 3-101 describes the `gxCapRecord` structure, which you use when retrieving or specifying join information. That section also describes what types of shapes you may use as join shapes.

You can use the `GXGetStyleJoin` function to retrieve the join information from a style object and the `GXSetStyleJoin` function to specify join information for a style object.

The `GXGetShapeJoin` and `GXSetShapeJoin` functions provide a way to retrieve and specify join information for the style object associated with a particular shape.

## GXGetStyleJoin

---

You can use the `GXGetStyleJoin` function to retrieve the join information from a style object.

```
gxJoinRecord *GXGetStyleJoin(gxStyle source, gxJoinRecord *join);
```

*source*        A reference to the style object whose join information you want to retrieve.

*join*            A pointer to a `gxJoinRecord` structure. On return, this structure contains the join information for the source style object.

*function result* A copy of the `gxJoinRecord` structure associated with the source style object.

### DESCRIPTION

The `GXGetStyleJoin` function returns as its function result, and in the `join` parameter, a pointer to a `gxJoinRecord` structure containing the join information for the style object specified by the `source` parameter.

This function creates a new shape to encapsulate the join geometry, and places a reference to this shape in the `join` field of the returned `gxJoinRecord` structure. You should dispose of this shape when you no longer need it.

Since this function copies the join information from the source style, you may make changes to the `gxJoinRecord` structure returned by this function without affecting the source style's join information. If you want to change the join information in the source style, you must use the `GXSetStyleJoin` function.

#### SPECIAL CONSIDERATIONS

Unless an error results, the `GXGetStyleJoin` function creates a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about disposing of QuickDraw GX objects.

#### ERRORS, WARNINGS, AND NOTICES

##### Errors

`out_of_memory`  
`style_is_nil`  
`parameter_is_nil`

#### SEE ALSO

For a discussion of joins, see “Joins” on page 3-25.

For examples of adding joins to shapes, see “Adding Joins to a Shape” on page 3-61 and “Adding Standard Joins to a Shape” on page 3-64.

For a discussion of the `gxJoinRecord` structure and a description of what types of shapes you can use as join shapes, see “The Join Structure” on page 3-101.

To specify join information for a style object, use the `GXSetStyleJoin` function, which is described in the next section.

To retrieve join information from a style object associated with a particular shape, use the `GXGetShapeJoin` function, which is described on page 3-132.

To specify join information for a style object associated with a particular shape, use the `GXSetShapeJoin` function, which is described on page 3-133.

## GXSetStyleJoin

---

You can use the `GXSetStyleJoin` function to change a style object's join information.

```
void GXSetStyleJoin(gxStyle target, const gxJoinRecord *join);
```

## Geometric Styles

<code>target</code>	A reference to the style object whose join information you want to change.
<code>join</code>	A pointer to the new join information.

## DESCRIPTION

The `GXSetStyleJoin` function replaces the join information in the style object specified by the `target` parameter with the join information specified in the `join` parameter. You use the `gxJoinRecord` structure to provide join information.

Passing `nil` for the `join` parameter indicates that you want no join shape and QuickDraw GX removes any join information from the target style.

When you set a style's join property using this function, you are effectively changing the joins for all shapes that share the style.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>style_is_nil</code>	
<code>parameter_out_of_range</code>	(debugging version)
<code>empty_shape_not_allowed</code>	(debugging version)
<code>ignorePlatformShape_not_allowed</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>nil_style_in_glyph_not_allowed</code>	(debugging version)
<code>complex_glyph_style_not_allowed</code>	(debugging version)
<code>shapeFill_not_allowed</code>	(debugging version)

**Notices (debugging version)**

<code>join_type_already_set</code>
<code>tags_in_shape_ignored</code>

## SEE ALSO

For a discussion of joins, see “Joins” on page 3-25.

For examples of adding joins to shapes, see “Adding Joins to a Shape” on page 3-61 and “Adding Standard Joins to a Shape” on page 3-64.

For a discussion of the `gxJoinRecord` structure and a description of what types of shapes you can use as join shapes, see “The Join Structure” on page 3-101.

To retrieve join information from a style object, use the `GXGetStyleJoin` function, which is described on page 3-129.

To retrieve join information from a style object associated with a particular shape, use the `GXGetShapeJoin` function, which is described in the next section.

To specify join information for a style object associated with a particular shape, use the `GXSetShapeJoin` function, which is described on page 3-133.

## GXGetShapeJoin

---

You can use the `GXGetShapeJoin` function to retrieve the join information from the style object of a shape.

```
gxJoinRecord *GXGetShapeJoin(gxShape source, gxJoinRecord *join);
```

`source`      A reference to the shape whose join information you want to retrieve.

`join`          A pointer to a `gxJoinRecord` structure. On return, this structure contains the join information for the source shape.

*function result*   A copy of the `gxJoinRecord` structure associated with the source shape's style object.

### DESCRIPTION

The `GXGetShapeJoin` function returns as its function result, and in the `join` parameter, a pointer to a `gxJoinRecord` structure containing the join information for the style object of the shape specified by the `source` parameter.

This function creates a new shape to encapsulate the join geometry, and places a reference to this shape in the `join` field of the returned `gxJoinRecord` structure. You should dispose of this shape when you no longer need it.

Since this function copies the join information from the source shape's style, you may make changes to the `gxJoinRecord` structure returned by this function without affecting the source shape's joins. If you want to change the join information for the source shape, you must use the `GXSetShapeJoin` function.

### SPECIAL CONSIDERATIONS

Unless an error results, the `GXGetShapeJoin` function creates a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about disposing of QuickDraw GX objects.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`

`shape_is_nil`

`parameter_is_nil`

**SEE ALSO**

For a discussion of joins, see “Joins” on page 3-25.

For examples of adding joins to shapes, see “Adding Joins to a Shape” on page 3-61 and “Adding Standard Joins to a Shape” on page 3-64.

For a discussion of the `gxJoinRecord` structure and a description of what types of shapes you can use as join shapes, see “The Join Structure” on page 3-101.

To retrieve join information from a style object, use the `GXGetStyleJoin` function, which is described on page 3-129. To specify join information for a style object, use the `GXSetStyleJoin` function, which is described on page 3-130.

To specify join information for a style object associated with a particular shape, use the `GXSetShapeJoin` function, which is described in the next section.

## **GXSetShapeJoin**

---

You can use the `GXSetShapeJoin` function to change the join information for the style object of a particular shape.

```
void GXSetShapeJoin(gxShape target, const gxJoinRecord *join);
```

<code>target</code>	A reference to the shape whose join information you want to change.
<code>join</code>	A pointer to new join information.

**DESCRIPTION**

The `GXSetShapeJoin` function replaces the join information in the style object of the shape specified by the `target` parameter with the join information provided in the `join` parameter. You use the `gxJoinRecord` structure to provide join information.

Passing `nil` for the `join` parameter indicates that you want no joins and QuickDraw GX removes any join information from the target shape.

If the target shape shares its style object with other shapes, this function makes a copy of the style object, sets the target shape to reference the copy, and changes the join property of the copy. (However, if the effect of this function would leave the join information unchanged, this function does not create a copy of the style object; instead, it posts a notice.)

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>style_is_nil</code>	
<code>parameter_out_of_range</code>	(debugging version)
<code>empty_shape_not_allowed</code>	(debugging version)
<code>ignorePlatformShape_not_allowed</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>nil_style_in_glyph_not_allowed</code>	(debugging version)
<code>complex_glyph_style_not_allowed</code>	(debugging version)
<code>shapeFill_not_allowed</code>	(debugging version)

**Notices (debugging version)**

`join_type_already_set`  
`tags_in_shape_ignored`

## SEE ALSO

For a discussion of joins, see “Joins” on page 3-25.

For examples of adding joins to shapes, see “Adding Joins to a Shape” on page 3-61 and “Adding Standard Joins to a Shape” on page 3-64.

For a discussion of the `gxJoinRecord` structure and a description of what types of shapes you can use as join shapes, see “The Join Structure” on page 3-101.

To retrieve join information from a style object, use the `GXGetStyleJoin` function, which is described on page 3-129. To specify join information for a style object, use the `GXSetStyleJoin` function, which is described on page 3-130.

To retrieve join information from a style object associated with a particular shape, use the `GXGetShapeJoin` function, which is described on page 3-132.

Getting and Setting Dashes

---

QuickDraw GX allows you to specify how contours of a shape should be dashed when drawn. In particular, you may specify a dash shape for any line, curve, rectangle, polygon, or path shape that has an open-frame shape fill or a closed-frame shape fill. You must always specify dash shapes in primitive form.

“The Dash Structure” on page 3-103 describes the `gxDashRecord` structure, which you use when retrieving or specifying dash information. That section also describes what types of shapes you may use as a dash shape.

You can use the `GXGetStyleDash` function to retrieve the dash information from a style object and the `GXSetStyleDash` function to specify dash information for a style object.

The `GXGetShapeDash` and `GXSetShapeDash` functions provide a way to retrieve and specify dash information for the style object associated with a particular shape.



## GXGetStyleDash

---

You can use the `GXGetStyleDash` function to retrieve the dash information from a style object.

```
gxDashRecord *GXGetStyleDash(gxStyle source, gxDashRecord *dash);
```

**source**      A reference to the style object whose dash information you want to retrieve.

**dash**          A pointer to a `gxDashRecord` structure. On return, this structure contains the dash information for the source style object.

**function result** A copy of the `gxDashRecord` structure associated with the source style object.

### DESCRIPTION

The `GXGetStyleDash` function returns as its function result, and in the `dash` parameter, a pointer to a `gxDashRecord` structure containing the dash information for the style object specified by the `source` parameter.

This function creates a new shape to encapsulate the dash geometry and places a reference to this shape in the `dash` field of the returned `gxDashRecord` structure. You should dispose of this shape when you no longer need it.

Since this function copies the dash information from the source style, you may make changes to the `gxDashRecord` structure returned by this function without affecting the source style's dash information. If you want to change the dash information in the source style, you must use the `GXSetStyleDash` function.

### SPECIAL CONSIDERATIONS

Unless an error results, the `GXGetStyleDash` function creates a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about disposing of QuickDraw GX objects.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

```
out_of_memory
style_is_nil
parameter_is_nil
```

**SEE ALSO**

For a discussion of dashes, see “Dashes” on page 3-27.

For examples of adding dashes to shapes, see page 3-66 through page 3-86.

For a discussion of the `gxDashRecord` structure and a description of what types of shapes you can use as dash shapes, see “The Dash Structure” on page 3-103.

To specify dash information for a style object, use the `GXSetStyleDash` function, which is described in the next section.

To retrieve dash information from a style object associated with a particular shape, use the `GXGetShapeDash` function, which is described on page 3-138. To specify dash information for a style object associated with a particular shape, use the `GXSetShapeDash` function, which is described on page 3-139.

To determine where dashing occurs for a particular shape, use the `GXGetShapeDashPositions` function, which is described on page 3-140.

## **GXSetStyleDash**

---

You can use the `GXSetStyleDash` function to change a style object’s dash information.

```
void GXSetStyleDash(gxStyle target, const gxDashRecord *dash);
```

<code>target</code>	A reference to the style object whose dash information you want to change.
<code>dash</code>	A pointer to the new dash information.

**DESCRIPTION**

The `GXSetStyleDash` function replaces the dash information in the style object specified by the `target` parameter with the dash information provided by the `dash` parameter. You use the `gxDashRecord` structure to provide dash information.

Passing `nil` for the `dash` parameter indicates that you want no dashing to occur and QuickDraw GX removes any dash information from the target style.

When you set a style’s dash property using this function, you are effectively changing the dashes for all shapes that share the style.

## ERRORS, WARNINGS, AND NOTICES

### Errors

<code>out_of_memory</code>	
<code>style_is_nil</code>	
<code>parameter_out_of_range</code>	(debugging version)
<code>empty_shape_not_allowed</code>	(debugging version)
<code>ignorePlatformShape_not_allowed</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>nil_style_in_glyph_not_allowed</code>	(debugging version)
<code>complex_glyph_style_not_allowed</code>	(debugging version)
<code>shapeFill_not_allowed</code>	(debugging version)

### Warnings

`graphic_type_cannot_be_dashed`

### Notices (debugging version)

`dash_already_set`  
`tags_in_shape_ignored`

## SEE ALSO

For a discussion of dashes, see “Dashes” on page 3-27.

For examples of adding dashes to shapes, see page 3-66 through page 3-86.

For a discussion of the `gxDashRecord` structure and a description of what types of shapes you can use as dash shapes, see “The Dash Structure” on page 3-103.

To retrieve dash information from a style object, use the `GXGetStyleDash` function, which is described on page 3-135.

To retrieve dash information from a style object associated with a particular shape, use the `GXGetShapeDash` function, which is described in the next section. To specify dash information for a style object associated with a particular shape, use the `GXSetShapeDash` function, which is described on page 3-139.

To determine where dashing occurs for a particular shape, use the `GXGetShapeDashPositions` function, which is described on page 3-140.

## GXGetShapeDash

---

You can use the `GXGetShapeDash` function to retrieve the dash information from the style object associated with a particular shape.

```
gxDashRecord *GXGetShapeDash(gxShape source, gxDashRecord *dash);
```

**source**      A reference to the shape whose dash information you want to retrieve.

**dash**      A pointer to a `gxDashRecord` structure. On return, this structure contains the dash information for the source shape.

**function result** A copy of the `gxDashRecord` structure associated with the source shape's style object.

### DESCRIPTION

The `GXGetShapeDash` function returns as its function result and in the `dash` parameter, a pointer to a `gxDashRecord` structure containing the dash information for the style object of the shape specified by the `source` parameter.

This function creates a new shape to encapsulate the dash geometry, and places a reference to this shape in the `dash` field of the returned `gxDashRecord` structure. You should dispose of this shape when you no longer need it.

Since this function copies the dash information from the source shape's style, you may make changes to the `gxDashRecord` structure returned by this function without affecting the source shape's dashes. If you want to change the dash information for the source shape, you must use the `GXSetShapeDash` function.

### SPECIAL CONSIDERATIONS

Unless an error results, the `GXGetShapeDash` function creates a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about disposing of objects.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`out_of_memory`

`shape_is_nil`

`parameter_is_nil`

**SEE ALSO**

For a discussion of dashes, see “Dashes” on page 3-27.

For examples of adding dashes to shapes, see page 3-66 through page 3-86.

For a discussion of the `gxDashRecord` structure and a description of what types of shapes you can use as dash shapes, see “The Dash Structure” on page 3-103.

To retrieve dash information from a style object, use the `GXGetStyleDash` function, which is described on page 3-135.

To specify dash information for a style object, use the `GXSetStyleDash` function, which is described on page 3-136.

To specify dash information for a style object associated with a particular shape, use the `GXSetShapeDash` function, which is described in the next section.

To determine where dashing occurs for a particular shape, use the `GXGetShapeDashPositions` function, which is described on page 3-140.

## **GXSetShapeDash**

---

You can use the `GXSetShapeDash` function to change the dash information for a style object associated with a particular shape.

```
void GXSetShapeDash(gxShape target, const gxDashRecord *dash);
```

`target`            A reference to the shape whose dash information you want to change.

`dash`             A pointer to the new dash information.

**DESCRIPTION**

The `GXSetShapeDash` function replaces the dash information in the style object of the shape specified by the `target` parameter with the dash information provided by the `dash` parameter. You use the `gxDashRecord` structure to provide dash information.

Passing `nil` for the `dash` parameter indicates that you want no dashing to occur and QuickDraw GX removes any dash information from the target shape.

If the target shape shares its style object with other shapes, this function makes a copy of the style object, sets the target shape to reference the copy, and changes the dash property of the copy. (However, if the effect of this function would leave the dash information unchanged, this function does not create a copy of the style object; instead, it returns a notice.)

**ERRORS, WARNINGS, AND NOTICES****Errors**

out_of_memory	
style_is_nil	
parameter_out_of_range	(debugging version)
empty_shape_not_allowed	(debugging version)
ignorePlatformShape_not_allowed	(debugging version)
illegal_type_for_shape	(debugging version)
nil_style_in_glyph_not_allowed	(debugging version)
complex_glyph_style_not_allowed	(debugging version)
shapeFill_not_allowed	(debugging version)

**Warnings**

graphic_type_cannot_be_dashed
-------------------------------

**Notices (debugging version)**

dash_already_set
tags_in_shape_ignored

**SEE ALSO**

For a discussion of dashes, see “Dashes” on page 3-27.

For examples of adding dashes to shapes, see page 3-66 through page 3-86.

For a discussion of the `gxDashRecord` structure and a description of what types of shapes you can use as dash shapes, see “The Dash Structure” on page 3-103.

To retrieve dash information from a style object, use the `GXGetStyleDash` function, which is described on page 3-135.

To specify dash information for a style object, use the `GXSetStyleDash` function, which is described on page 3-136.

To retrieve dash information from a style object associated with a particular shape, use the `GXGetShapeDash` function, which is described on page 3-138.

To determine where dashing occurs for a particular shape, use the `GXGetShapeDashPositions` function, which is described in the next section.

## **GXGetShapeDashPositions**

---

You can use the `GXGetShapeDashPositions` function to determine the precise locations where QuickDraw GX draws a particular shape’s dashes.

```
long GXGetShapeDashPositions(gxShape source,
                             gxMapping dashMappings[]);
```

## Geometric Styles

**source** A reference to the shape whose dash positions you want to find.

**dashMappings** An array of dash positions. On return, this array contains mappings that indicate the position of the dashes of the source shape.

**function result** The number of dash positions returned in the `dashMappings` parameter.

**DESCRIPTION**

The `GXGetShapeDashPositions` function returns in the `dashMappings` parameter mappings that indicate the locations and rotations of the dashes as drawn along the contours of the source shape.

The function result is the number of dash positions returned—the number of dash shapes drawn along the contours of the source shape.

If you pass `nil` for the `dashMappings` parameter, the `GXGetShapeDashPositions` function still returns as the function result the number of dashes but it does not return the positions of the dashes.

This function returns 0 if the source shape is not dashed.

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`  
`shape_is_nil`

**Warnings**

`graphic_type_cannot_be_dashed`

**SEE ALSO**

For a discussion of dashes, see “Dashes” on page 3-27.

For an example of using this function, see “Determining Dash Positions” on page 3-81.

For a discussion of the `gxDashRecord` structure, see “The Dash Structure” on page 3-103.

To retrieve dash information from a style object, use the `GXGetStyleDash` function, which is described on page 3-135. To specify dash information for a style object, use the `GXSetStyleDash` function, which is described on page 3-136.

To retrieve dash information from a style object associated with a particular shape, use the `GXGetShapeDash` function, which is described on page 3-138. To specify dash information for a style object associated with a particular shape, use the `GXSetShapeDash` function, which is described on page 3-139.

## Getting and Setting Patterns

---

QuickDraw GX allows you to specify a pattern to fill a shape when drawn. In particular, you may specify a pattern shape for any line, curve, rectangle, polygon, or path shape that has any framed shape fill or any solid fill. You must always specify pattern shapes in their primitive form.

“The Pattern Structure” on page 3-106 describes the `gxPatternRecord` structure, which you use when retrieving or specifying pattern information. That section also describes what types of shapes you may use as a pattern shape.

You can use the `GXGetStylePattern` function to retrieve the pattern information from a style object and the `GXSetStylePattern` function to specify pattern information for a style object.

The `GXGetShapePattern` and `GXSetShapePattern` functions provide a way to retrieve and specify pattern information for the style object associated with a particular shape.

## GXGetStylePattern

---

You can use the `GXGetStylePattern` function to retrieve the pattern information from a style object.

```
gxPatternRecord *GXGetStylePattern(gxStyle source,
                                   gxPatternRecord *pattern);
```

**source**      The style object whose pattern information you want to retrieve.

**pattern**      A pointer to a `gxPatternRecord` structure. On return, this structure contains the pattern information for the source style object.

*function result*    A copy of the `gxPatternRecord` structure associated with the source style object.

### DESCRIPTION

The `GXGetStylePattern` function returns as its function result, and in the `pattern` parameter, a pointer to a `gxPatternRecord` structure containing the pattern information for the style object specified by the `source` parameter.



This function creates a new shape to encapsulate the pattern geometry, and places a reference to this shape in the `pattern` field of the returned `gxPatternRecord` structure. You should dispose of this shape when you no longer need it.

Since this function copies the pattern information from the source style, you may make changes to the `gxPatternRecord` structure returned by this function without affecting the source style's pattern information. If you want to change the pattern information in the source style, you must use the `GXSetStylePattern` function.

#### SPECIAL CONSIDERATIONS

Unless an error results, the `GXGetStylePattern` function creates a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about disposing of QuickDraw GX objects.

#### ERRORS, WARNINGS, AND NOTICES

##### Errors

`out_of_memory`  
`style_is_nil`  
`parameter_is_nil`

#### SEE ALSO

For a discussion of patterns, see “Patterns” on page 3-31.

For examples of adding patterns to shapes, see page 3-86 through page 3-91.

For a discussion of the `gxPatternRecord` structure and a description of what types of shapes you can use as pattern shapes, see “The Pattern Structure” on page 3-106.

To specify pattern information for a style object, use the `GXSetStylePattern` function, which is described in the next section.

To retrieve pattern information from a style object associated with a particular shape, use the `GXGetShapePattern` function, which is described on page 3-145. To specify pattern information for a style object associated with a particular shape, use the `GXSetShapePattern` function, which is described on page 3-146.

To determine where pattern shapes are drawn for a particular shape, use the `GXGetShapePatternPositions` function, which is described on page 3-147.

## GXSetStylePattern

---

You can use the `GXSetStylePattern` function to change a style object's pattern information.

```
void GXSetStylePattern(gxStyle target,
                      const gxPatternRecord *pattern);
```

**target**        A reference to the style object whose pattern information you want to change.

**pattern**       A pointer to the new pattern information.

### DESCRIPTION

The `GXSetStylePattern` function replaces the pattern information in the style object specified by the `target` parameter with the pattern information provided by the `pattern` parameter. You use the `gxPatternRecord` structure to provide pattern information.

Passing `nil` for the `pattern` parameter indicates that you want no pattern and QuickDraw GX removes any pattern information from the target style.

When you set a style's pattern property using this function, you are effectively changing the pattern for all shapes that share the style.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

<code>out_of_memory</code>	
<code>style_is_nil</code>	
<code>parameter_out_of_range</code>	(debugging version)
<code>empty_shape_not_allowed</code>	(debugging version)
<code>ignorePlatformShape_not_allowed</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>nil_style_in_glyph_not_allowed</code>	(debugging version)
<code>complex_glyph_style_not_allowed</code>	(debugging version)
<code>shapeFill_not_allowed</code>	(debugging version)
<code>colorProfile_must_be_nil</code>	(debugging version)

#### Warnings

`graphic_type_cannot_be_dashed`

#### Notices (debugging version)

`dash_already_set`  
`tags_in_shape_ignored`

**SEE ALSO**

For a discussion of patterns, see “Patterns” on page 3-31.

For examples of adding patterns to shapes, see page 3-86 through page 3-91.

For a discussion of the `gxPatternRecord` structure and a description of what types of shapes you can use as pattern shapes, see “The Pattern Structure” on page 3-106.

To retrieve pattern information from a style object, use the `GXGetStylePattern` function, which is described on page 3-142.

To retrieve pattern information from a style object associated with a particular shape, use the `GXGetShapePattern` function, which is described in the next section. To specify pattern information for a style object associated with a particular shape, use the `GXSetShapePattern` function, which is described on page 3-146.

To determine where pattern shapes are drawn for a particular shape, use the `GXGetShapePatternPositions` function, which is described on page 3-147.

## **GXGetShapePattern**

---

You can use the `GXGetShapePattern` function to retrieve the pattern information from the style object associated with a particular shape.

```
gxPatternRecord *GXGetShapePattern(gxShape source,
                                   gxPatternRecord *pattern);
```

<code>source</code>	The shape whose pattern information you want to retrieve.
<code>pattern</code>	A pointer to a <code>gxPatternRecord</code> structure. On return, this structure contains the pattern information for the source shape.

<i>function result</i>	A copy of the <code>gxPatternRecord</code> structure associated with the source shape's style object.
------------------------	---

**DESCRIPTION**

The `GXGetShapePattern` function returns as its function result and in the `pattern` parameter a pointer to a `gxPatternRecord` structure containing the pattern information for the style object of the shape specified by the `source` parameter.

This function creates a new shape to encapsulate the pattern geometry, and places a reference to this shape in the `pattern` field of the returned `gxPatternRecord` structure. You should dispose of this shape when you no longer need it.

Since this function copies the pattern information from the source shape's style, you may make changes to the `gxPatternRecord` structure returned by this function without affecting the source shape's pattern. If you want to change the pattern information for the source shape, you must use the `GXSetShapePattern` function.

**SPECIAL CONSIDERATIONS**

The `GXGetShapePattern` function may create a shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about creating and disposing of objects.

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`  
`shape_is_nil`  
`parameter_is_nil`

**SEE ALSO**

For a discussion of patterns, see “Patterns” on page 3-31.

For examples of adding patterns to shapes, see page 3-86 through page 3-91.

For a discussion of the `gxPatternRecord` structure and a description of what types of shapes you can use as pattern shapes, see “The Pattern Structure” on page 3-106.

To retrieve pattern information from a style object, use the `GXGetStylePattern` function, which is described on page 3-142. To specify pattern information for a style object, use the `GXSetStylePattern` function, which is described on page 3-144.

To specify pattern information for a style object associated with a particular shape, use the `GXSetShapePattern` function, which is described in the next section.

To determine where pattern shapes are drawn for a particular shape, use the `GXGetShapePatternPositions` function, which is described on page 3-147.

**GXSetShapePattern**

---

You can use the `GXSetShapePattern` function to change the pattern information for a style object associated with a particular shape.

```
void GXSetShapePattern(gxShape target,
                      const gxPatternRecord *pattern);
```

`target`        A reference to the shape whose pattern information you want to change.  
`pattern`       A pointer to the new pattern information.

**DESCRIPTION**

The `GXSetShapePattern` function replaces the pattern information in the style object of the shape specified by the `target` parameter with the pattern information provided by the `pattern` parameter. You use the `gxPatternRecord` structure to provide pattern information.

Passing `nil` for the `pattern` parameter indicates that you want no pattern and QuickDraw GX removes any pattern information from the target shape.

If the `target` shape shares its style object with other shapes, this function makes a copy of the style object, sets the target shape to reference the copy, and changes the `pattern` property of the copy. (However, if the effect of this function would leave the pattern information unchanged, this function does not create a copy of the style object; instead, it returns a notice).

#### ERRORS, WARNINGS, AND NOTICES

##### Errors

`out_of_memory`  
`shape_is_nil`  
`parameter_out_of_range`  
`pattern_lattice_out_of_range`

##### Notices (debugging version)

`pattern_already_set`

#### SEE ALSO

For a discussion of patterns, see “Patterns” on page 3-31.

For examples of adding patterns to shapes, see page 3-86 through page 3-91.

For a discussion of the `gxPatternRecord` structure and a description of what types of shapes you can use as pattern shapes, see “The Pattern Structure” on page 3-106.

To retrieve pattern information from a style object, use the `GXGetStylePattern` function, which is described on page 3-142. To specify pattern information for a style object, use the `GXSetStylePattern` function, which is described on page 3-144.

To retrieve pattern information from a style object associated with a particular shape, use the `GXGetShapePattern` function, which is described on page 3-145.

To determine where pattern shapes are drawn for a particular shape, use the `GXGetShapePatternPositions` function, which is described in the next section.

## GXGetShapePatternPositions

---

You can use the `GXGetShapePatternPositions` function to determine the precise locations where QuickDraw GX draws the shapes that pattern another shape.

```
long GXGetShapePatternPositions(gxShape source,
                               gxPoint positions[]);
```

## Geometric Styles

**source** A reference to the shape whose pattern positions you want to find.

**positions** An array of pattern positions. On return, this array contains points that indicate the position of the pattern shapes that pattern the source shape.

**function result** The number of pattern positions returned in the `positions` parameter.

**DESCRIPTION**

The `GXGetShapePatternPositions` function returns in the `positions` parameter the locations of the pattern shapes as drawn for the source shape.

The function result is the number of pattern positions returned—the number of pattern shapes drawn for the source shape.

If you pass `nil` for the `positions` parameter, the `GXGetShapePatternPositions` function still returns as the function result the number of pattern shapes but it does not return the positions of the pattern shapes.

This function returns 0 if the source shape has no pattern.

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`  
`shape_is_nil`

**SEE ALSO**

For a discussion of patterns, see “Patterns” on page 3-31.

For an example using this function, see “Determining Pattern Positions” on page 3-88.

For a discussion of the `gxPatternRecord` structure and a description of what types of shapes you can use as pattern shapes, see “The Pattern Structure” on page 3-106.

To retrieve pattern information from a style object, use the `GXGetStylePattern` function, which is described on page 3-142. To specify pattern information for a style object, use the `GXSetStylePattern` function, which is described on page 3-144.

To retrieve pattern information from a style object associated with a particular shape, use the `GXGetShapePattern` function, which is described on page 3-145. To specify pattern information for a style object associated with a particular shape, use the `GXSetShapePattern` function, which is described on page 3-146.

## Summary of Geometric Styles

---

### Constants and Data Types

---

#### Style Attributes

```
enum gxStyleAttributes {
    gxCenterFrameStyle    = 0,           /* center the pen on contour */
    gxSourceGridStyle     = 0x0001,      /* constrain to source grid */
    gxDeviceGridStyle     = 0x0002,      /* constrain to device grid */
    gxInsideFrameStyle    = 0x0004,      /* place pen inside contour */
    gxOutsideFrameStyle   = 0x0008,      /* place pen outside contour */
    gxAutoInsetStyle      = 0x0010      /* don't assume right is in */
};

typedef long gxStyleAttribute;
```

#### Cap Structure

```
struct gxCapRecord {
    gxCapAttribute attributes; /* modifies behavior of caps */
    gxShape          startCap; /* shape to use at start of contours */
    gxShape          endCap;   /* shape to use at end of contours */
};
```

#### Cap Attributes

```
enum gxCapAttributes {
    gxLevelStartCap= 0x0001; /* suppress start cap rotation */
    gxLevelEndCap  = 0x0002; /* suppress end cap rotation */
};

typedef long gxCapAttribute;
```

#### Join Structure

```
struct gxJoinRecord {
    gxJoinAttribute attributes; /* modifies behavior of joins */
    gxShape          join;      /* shape to use at corners */
    Fixed            miter;     /* size limit for sharp joins */
};
```

**Join Attributes**

```
enum gxJoinAttributes {
    gxSharpJoin = 0x0000,    /* use default sharp joins */
    gxCurveJoin = 0x0001,    /* use default curved joins */
    gxLevelJoin = 0x0002     /* suppress join shape rotation */
};

typedef long gxJoinAttribute;
```

**Dash Structure**

```
struct gxDashRecord {
    gxDashAttribute attributes; /* modifies behavior of dashes */
    gxShape          dash;      /* shape used for dashing */
    Fixed            advance;    /* distance between dashes */
    fract            phase;      /* start offset into the contour */
    Fixed            scale;      /* height of dash (mapped to pen) */
};
```

**Dash Attributes**

```
typedef enum gxDashAttributes {
    gxBendDash      = 0x0001;    /* distorts shape in 1 dimension */
    gxBreakDash     = 0x0002;    /* places dash contours separately */
    gxClipDash      = 0x0004;    /* clips dashes to pen width */
    gxLevelDash     = 0x0008;    /* suppresses dash rotation */
    gxAutoAdvanceDash = 0x0010;  /* automatically adjusts advances */
};

typedef long gxDashAttribute;
```

**Pattern Structure**

```
struct gxPatternRecord {
    gxPatternAttribute attributes; /* modifies behavior of pattern */
    gxShape            pattern;    /* shape to use as pattern */
    gxPoint            u;          /* vector for pattern grid */
    gxPoint            v;          /* vector for pattern grid */
};
```



**Pattern Attributes**

```
enum gxPatternAttributes {
    gxPortAlignPattern = 0x0001, /* align pattern with device */
    gxPortMapPattern   = 0x0002 /* suppress mapping of pattern */
};

typedef long gxPatternAttribute;
```

**Functions for Manipulating Geometric Style Properties**

---

**Getting and Setting Style Attributes**

```
gxStyleAttribute GXGetStyleAttributes
                                (gxStyle source);
void GXSetStyleAttributes      (gxStyle target, gxStyleAttribute attributes);
gxStyleAttribute GXGetShapeStyleAttributes
                                (gxShape source);
void GXSetShapeStyleAttributes
                                (gxShape target, gxStyleAttribute attributes);
```

**Getting and Setting Curve Error**

```
Fixed GXGetStyleCurveError      (gxStyle source);
void GXSetStyleCurveError       (gxStyle target, Fixed error);
Fixed GXGetShapeCurveError      (gxShape source);
void GXSetShapeCurveError       (gxShape target, Fixed error);
```

**Getting and Setting the Pen Width**

```
Fixed GXGetStylePen             (gxStyle source);
void GXSetStylePen              (gxStyle target, Fixed pen);
Fixed GXGetShapePen             (gxShape source);
void GXSetShapePen              (gxShape target, Fixed pen);
```

**Getting and Setting Caps**

```
gxCapRecord *GXGetStyleCap      (gxStyle source, gxCapRecord *cap);
void GXSetStyleCap              (gxStyle target, const gxCapRecord *cap);
gxCapRecord *GXGetShapeCap      (gxShape source, gxCapRecord *cap);
void GXSetShapeCap              (gxShape target, const gxCapRecord *cap);
```

**Getting and Setting Joins**

```

gxJoinRecord *GXGetStyleJoin
                                (gxStyle source, gxJoinRecord *join);
void GXSetStyleJoin             (gxStyle target, const gxJoinRecord *join);
gxJoinRecord *GXGetShapeJoin
                                (gxShape source, gxJoinRecord *join);
void GXSetShapeJoin            (gxShape target, const gxJoinRecord *join);

```

**Getting and Setting Dashes**

```

gxDashRecord *GXGetStyleDash
                                (gxStyle source, gxDashRecord *dash);
void GXSetStyleDash            (gxStyle target, const gxDashRecord *dash);
gxDashRecord *GXGetShapeDash
                                (gxShape source, gxDashRecord *dash);
void GXSetShapeDash            (gxShape target, const gxDashRecord *dash);
long GXGetShapeDashPositions
                                (gxShape source, gxMapping dashMappings[]);

```

**Getting and Setting Patterns**

```

gxPatternRecord *GXGetStylePattern
                                (gxStyle source, gxPatternRecord *pattern);
void GXSetStylePattern         (gxStyle target, const gxPatternRecord
                                *pattern);
gxPatternRecord *GXGetShapePattern
                                (gxShape source,
                                gxPatternRecord *pattern);
void GXSetShapePattern         (gxShape target, const gxPatternRecord
                                *pattern);
long GXGetShapePatternPositions
                                (gxShape source, gxPoint positions[]);

```

# Geometric Operations

---

## Contents

About Geometric Operations	4-4
Contours and Contour Direction	4-4
Reducing and Simplifying Shape Geometries	4-9
The Primitive Form of Shape Geometries	4-12
Geometric Information	4-16
Touching and Containing	4-18
Geometric Arithmetic	4-21
Using Geometric Operations	4-23
Determining and Reversing Contour Direction	4-23
Breaking Shape Contours	4-28
Eliminating Unnecessary Geometric Points	4-30
Simplifying Shapes	4-33
Converting a Shape to Primitive Form	4-38
Finding Geometric Information About a Shape	4-41
Finding the Length of a Contour	4-42
Finding the Point at a Certain Distance Along a Contour	4-42
Finding the Bounding Rectangle and Center Point of a Shape	4-43
Finding the Area of a Shape	4-45
Setting a Shape's Bounding Rectangle	4-47
Insetting Shapes	4-50
Determining Whether Two Shapes Touch	4-53
Determining Whether One Shape Contains Another	4-58
Performing Geometric Arithmetic With Shapes	4-60
Geometric Operations Reference	4-67
Constants and Data Types	4-67
Contour Directions	4-67
Functions	4-68
Determining and Reversing Contour Direction	4-68
GXGetShapeDirection	4-68
GXReverseShape	4-70

<b>Breaking Shape Contours</b>	4-72
GXBreakShape	4-72
<b>Reducing and Simplifying Shapes</b>	4-74
GXReduceShape	4-74
GXSimplifyShape	4-76
<b>Incorporating Style Information Into Shape Geometries</b>	4-79
GXPrimitiveShape	4-79
<b>Finding Geometric Information About Shapes</b>	4-83
GXGetShapeLength	4-83
GXShapeLengthToPoint	4-85
GXGetShapeCenter	4-87
GXGetShapeArea	4-88
<b>Getting and Setting Shape Bounds</b>	4-90
GXGetShapeBounds	4-90
GXSetShapeBounds	4-92
<b>Insetting Shapes</b>	4-94
GXInsetShape	4-94
<b>Determining Whether Two Areas Touch</b>	4-95
GXTouchesRectanglePoint	4-96
GXTouchesBoundsShape	4-97
GXTouchesShape	4-98
<b>Determining Whether One Shape Contains Another</b>	4-100
GXContainsRectangle	4-100
GXContainsBoundsShape	4-101
GXContainsShape	4-103
<b>Performing Geometric Arithmetic With Shapes</b>	4-104
GXIntersectRectangle	4-105
GXUnionRectangle	4-106
GXIntersectShape	4-107
GXUnionShape	4-109
GXDifferenceShape	4-110
GXReverseDifferenceShape	4-112
GXExcludeShape	4-114
GXInvertShape	4-116
<b>Summary of Geometric Operations</b>	4-117
<b>Constants and Data Types</b>	4-117
<b>Functions</b>	4-117

## Geometric Operations

This chapter describes the functions that allow you to perform geometric operations on shapes. Some of the geometric operations described in this chapter work on all types of shapes. Read this chapter if you perform any kind of geometric manipulation on the shapes you create.

Before reading this chapter, you should be familiar with the QuickDraw GX object architecture as described in *Inside Macintosh: QuickDraw GX Objects*. You should also be familiar with the information in the chapters “Geometric Shapes” and “Geometric Styles” in this book.

For more information about geometric manipulation of shapes, you might want to read the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects* and the chapter “QuickDraw GX Mathematics” in *Inside Macintosh: QuickDraw GX Environment and Utilities*.

This chapter introduces the basic categories of geometric operations and shows how to use these operations to

- n determine and reverse the contour direction of a shape’s contours
- n simplify the geometric description of a shape
- n incorporate style information into a shape’s geometry
- n obtain geometric information about a shape’s geometry, such as contour length and area
- n determine and alter the bounding rectangle of a shape
- n inset a shape’s geometry
- n determine if two shapes touch
- n determine if one shape contains another
- n perform geometric arithmetic, such as intersection and union, on shapes

Finally, this chapter contains a complete reference for the geometric operations.

## About Geometric Operations

---

The geometric operations allow you to obtain geometric information about geometric shapes and perform geometric calculations on them without having to manipulate shape geometries directly.

The geometric operations fall into five main categories:

- n operations that affect contours and contour direction
- n operations that simplify the drawing of shapes
- n operations that determine and alter basic geometric information about shapes
- n operations that test for intersection and inclusion
- n operations that perform geometric arithmetic on shapes

The next five sections discuss these categories.

### Contours and Contour Direction

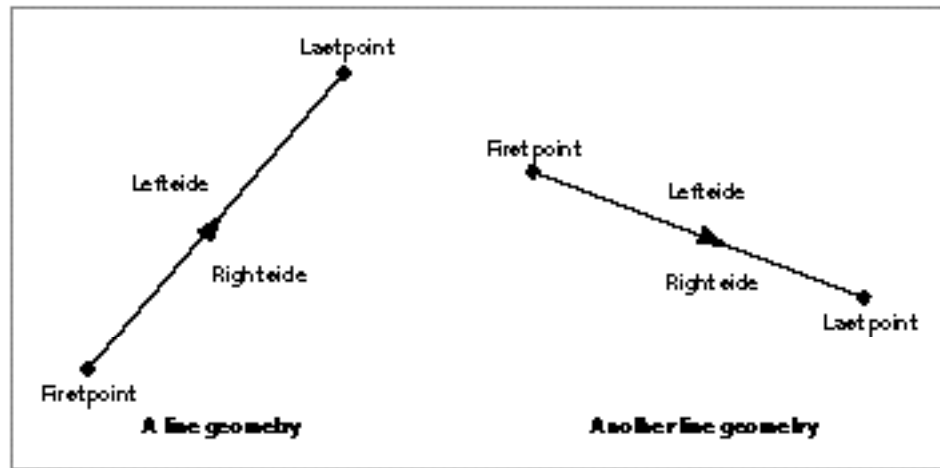
---

With the exception of empty, full, and point shapes, geometric shapes are made up of contours. Line, curve, and rectangle shapes have a single contour, while polygon and path shapes can have zero, one, or more contours. Every contour is defined by an ordered series of on-curve or off-curve geometric points, or a combination of both.

## Geometric Operations

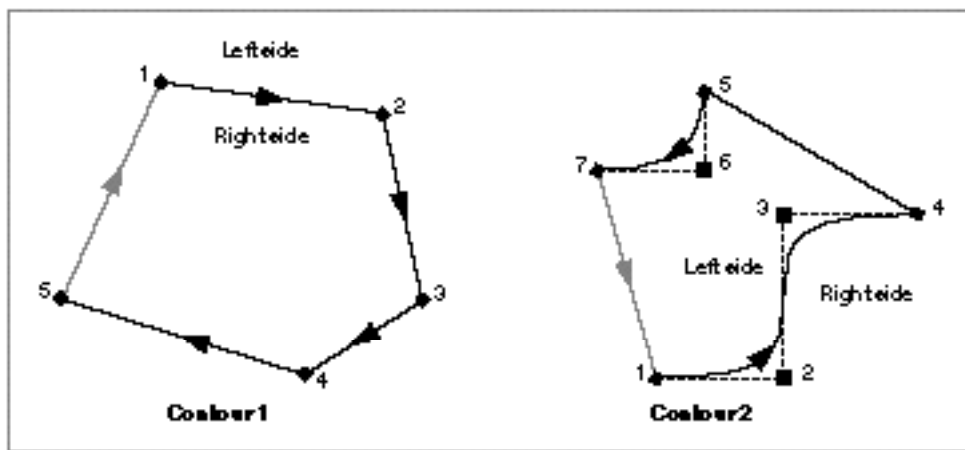
For example, the geometry of a line shape contains two (on-curve) geometric points—a first point and a last point. The contour of a line shape is the line segment connecting these two points. Since the line has a first point and a last point, it also has a direction, a right side, and a left side, as shown in Figure 4-1.

**Figure 4-1** Line contours



As another example, a path shape can have multiple contours; each path contour is defined by a series of on-curve and off-curve points. As with line contours, each path contour has a direction, a right side, and a left side. Notice that the order of the geometric points decides which side is the left side and which side is the right side, as shown in Figure 4-2.

**Figure 4-2** A path shape with two contours



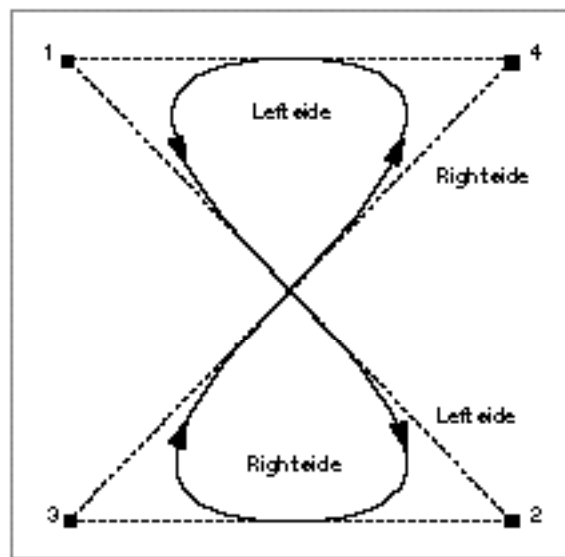
Each contour of a polygon or path shape has an implied line (or curve) connecting the last geometric point of the contour to the first geometric point of the contour. QuickDraw GX uses this implied line (or curve) when the shape fill of the polygon or path shape is the closed-frame shape fill or any of the solid shape fills. These implied lines are shown in gray in Figure 4-2.

Notice that the right side of the first contour falls inside the area enclosed by the contour and the right side of the second contour falls outside the area enclosed by the contour.



All contours have either a clockwise or a counterclockwise **contour direction**. Sometimes the contour direction of a contour is obvious, such as the contour directions of the contours in Figure 4-2. In this figure, the first contour has a clockwise contour direction and the second contour has a counterclockwise contour direction. However, sometimes the contour direction is not so obvious. Figure 4-3 gives an example.

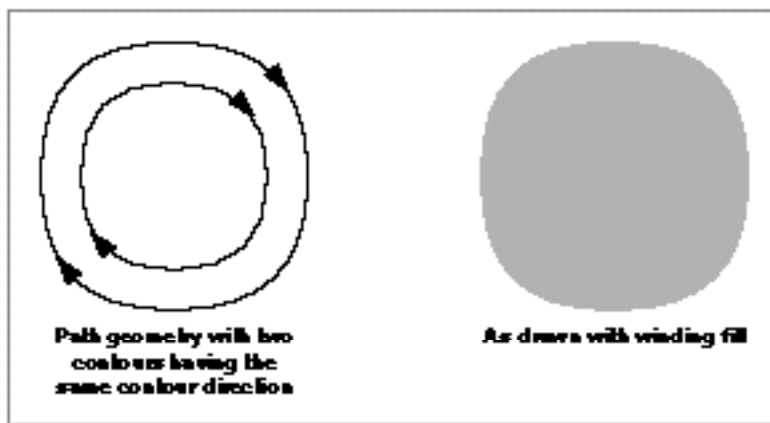
**Figure 4-3** A path whose contour direction is not immediately obvious



The upper half of the contour shown in Figure 4-3 seems to have a counterclockwise direction while the lower half of the contour seems to have a clockwise direction. In cases like this one, QuickDraw GX assigns an arbitrary contour direction to the entire contour. You can use the `GXGetShapeDirection` function, described on page 4-68, to find the contour direction that QuickDraw GX has assigned to a particular contour.

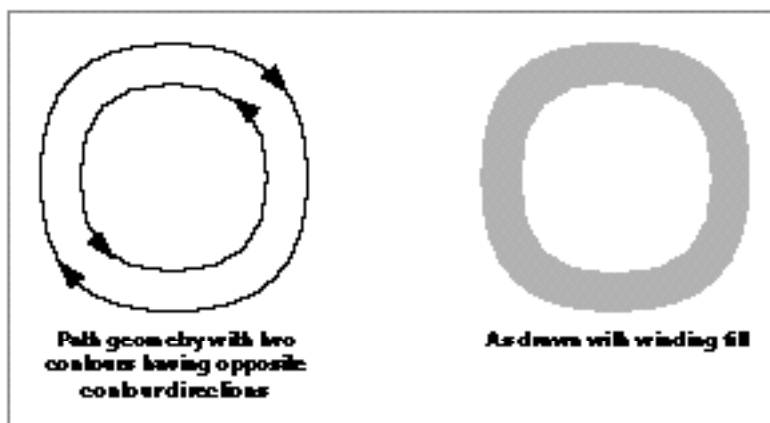
QuickDraw GX uses contour direction for a number of purposes—for example, when filling shapes that have a winding shape fill. The path shape shown in Figure 4-4 contains an inner contour with the same contour direction as the contour that surrounds it. When drawing this path using a winding fill, QuickDraw GX ignores the inner contour.

**Figure 4-4** A path whose inner contour has the same contour direction as its outer contour



To indicate that QuickDraw GX should not ignore the inner contour, you could change the shape fill to even-odd fill, or you could reverse the contour direction of the inner contour, as shown in Figure 4-5.

**Figure 4-5** A path shape whose inner and outer contours have different contour directions



QuickDraw GX lets you reverse a contour's direction by reversing the order of the geometric points in the contour.

**Note**

QuickDraw GX always considers line shapes to have a clockwise contour direction, regardless of the order of the geometric points in the line's geometry. Therefore, you cannot change the contour direction of line shapes. However, a line contour in a polygon or a path does have a clockwise or a counterclockwise direction (which QuickDraw GX assigns to it depending on the other contours in the shape); therefore, you can change the contour direction of line contours in polygons and paths. <sup>u</sup>

In certain situations, QuickDraw GX needs to know which side of a contour is the inside and which is the outside—for example, when drawing a geometric shape that has the inside-frame style attribute set. The default assumption is that the right side of a contour is the inside—which works well for clockwise contours but can produce surprising results with counterclockwise contours. The auto-inset style attribute indicates that QuickDraw GX should find the true inside for each contour of a shape, rather than assuming the right side is the inside. The **true inside** of a contour is defined to be the right side of the contour if the contour direction is clockwise and the left side of a contour if the contour direction is counterclockwise.

You can find more information about the inside-frame style attribute and the auto-inset style attribute in Chapter 3, “Geometric Styles,” in this book.

The section “Determining and Reversing Contour Direction” beginning on page 4-23 contains programming examples relating to contour direction.

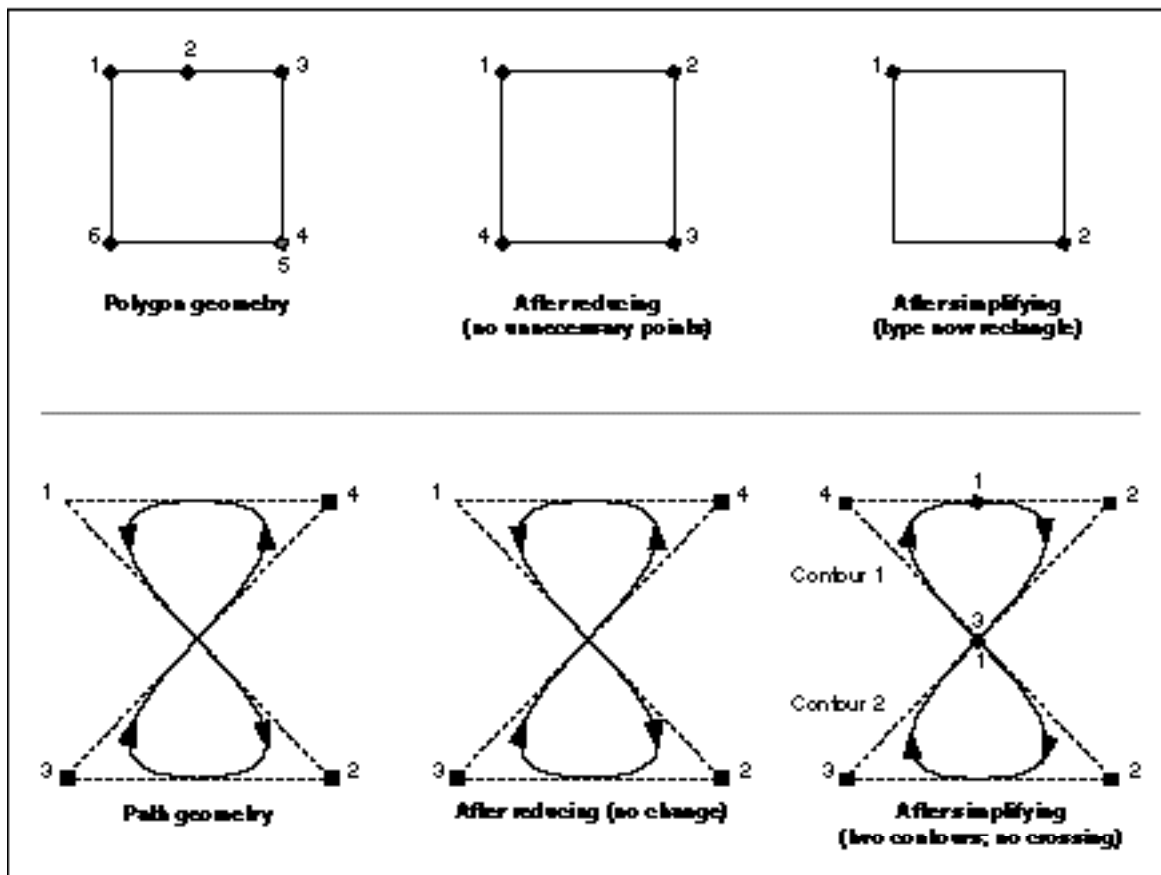
## Reducing and Simplifying Shape Geometries

---

QuickDraw GX allows you to change shape geometries to simpler forms. You can reduce the number of geometric points in a shape by removing unnecessary ones. You can also simplify a shape's geometry by removing unnecessary contour breaks, eliminating crossed and overlapping contours, and even simplifying the shape's shape type, if possible.

Figure 4-6 shows the difference between reducing a shape's geometry and simplifying a shape's geometry.

**Figure 4-6** Effects of reducing and simplifying shape geometries



In this figure, the polygon geometry has two unnecessary geometric points, which are removed in the reduced polygon. Since the polygon is actually a square, simplifying this polygon converts the polygon geometry to the simplest type of geometry necessary—which in this case is a rectangle geometry.

The path geometry in the lower part of Figure 4-6 has a crossed contour, but no unnecessary geometric points. Reducing this path results in the same path geometry, whereas simplifying this path reorders the geometric points and breaks the geometry into two path contours so that no contour crossing occurs. Also notice that, because the original path geometry starts with an off-curve control point, simplifying the path adds an initial on-curve geometric point. The new initial geometric point is halfway between the point that was originally at the end of the contour and the point that was originally at the beginning of the contour. Although adding this new complexity might not seem like a simplification, removing crossed contours does result in more predictable drawing results, as shown in Figure 4-7.

**Figure 4-7** How simplifying a shape can produce more predictable results when drawing

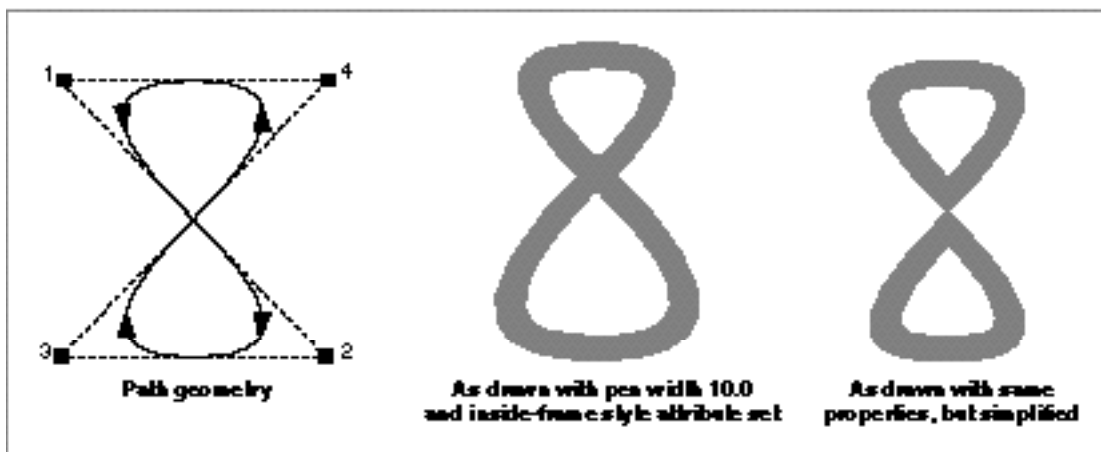


Figure 4-7 shows the path geometry from Figure 4-6. When this path is drawn with a pen width of 10.0 and the inside-frame style attribute set, the upper half of the path is inset, but the lower half of the path is outset, because of the crossed contour. Simplifying the shape uncrosses the contour, which results in both halves of the path shape being inset when drawn.

For more examples of the effect of simplifying shapes on drawing, see the section “Simplifying Shapes” beginning on page 4-33, as well as in the pen placement examples in Chapter 3, “Geometric Styles,” in this book.

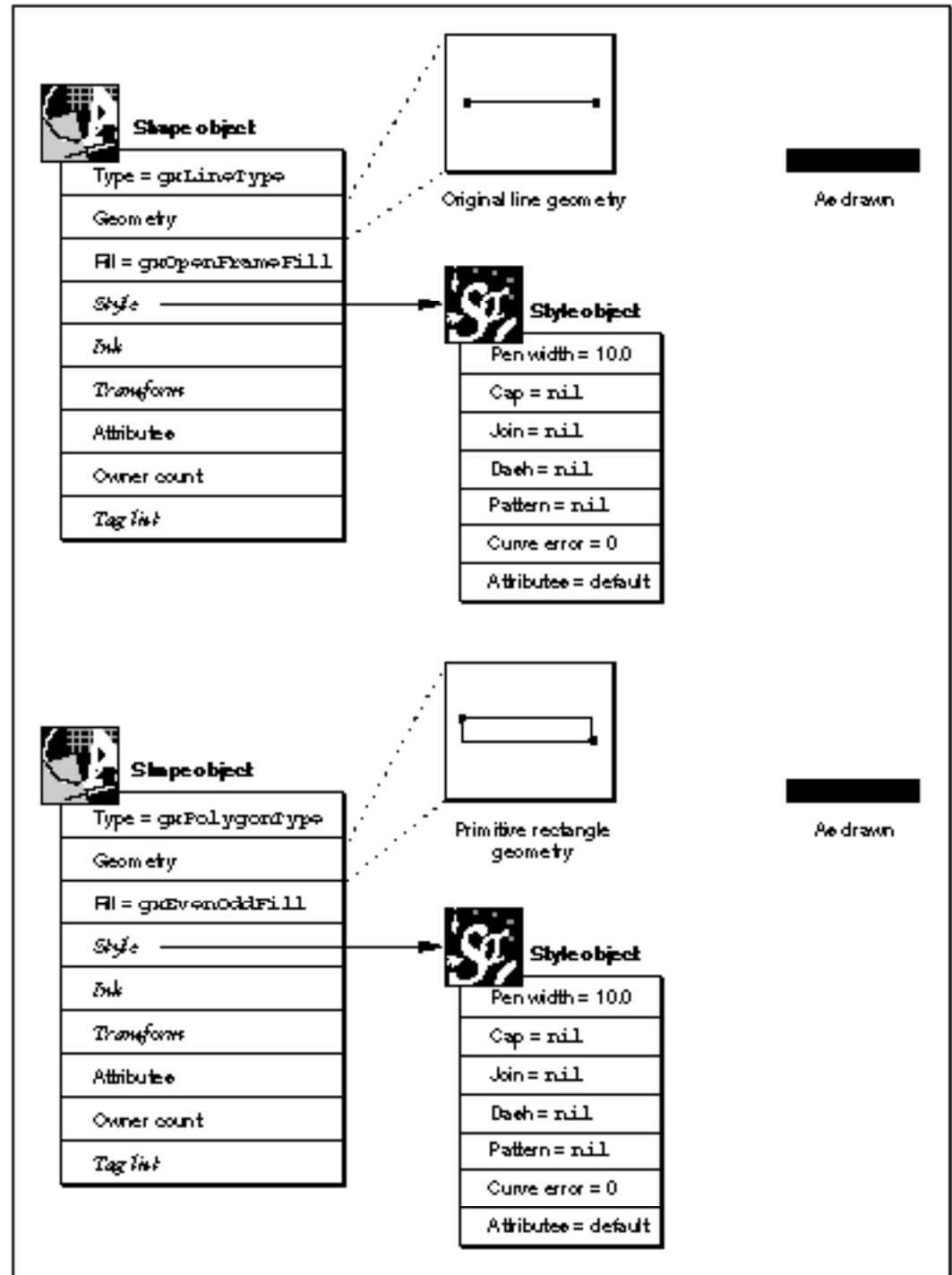
## The Primitive Form of Shape Geometries

---

QuickDraw GX provides a mechanism for incorporating the stylistic variations contained in a style object directly into the geometry of a shape object. This mechanism is the `GXPrimitiveShape` function. When the geometry of a shape has its stylistic variations incorporated into it, it is said to be in **primitive form**. Shapes in primitive form include

- n empty shapes and full shapes, which are described in Chapter 2, “Geometric Shapes”
- n filled rectangle, polygon, and path shapes, which are also described in Chapter 2, “Geometric Shapes”
- n hairline framed shapes, which are described in Chapter 3, “Geometric Styles”
- n glyph shapes, which are described in *Inside Macintosh: QuickDraw GX Typography*

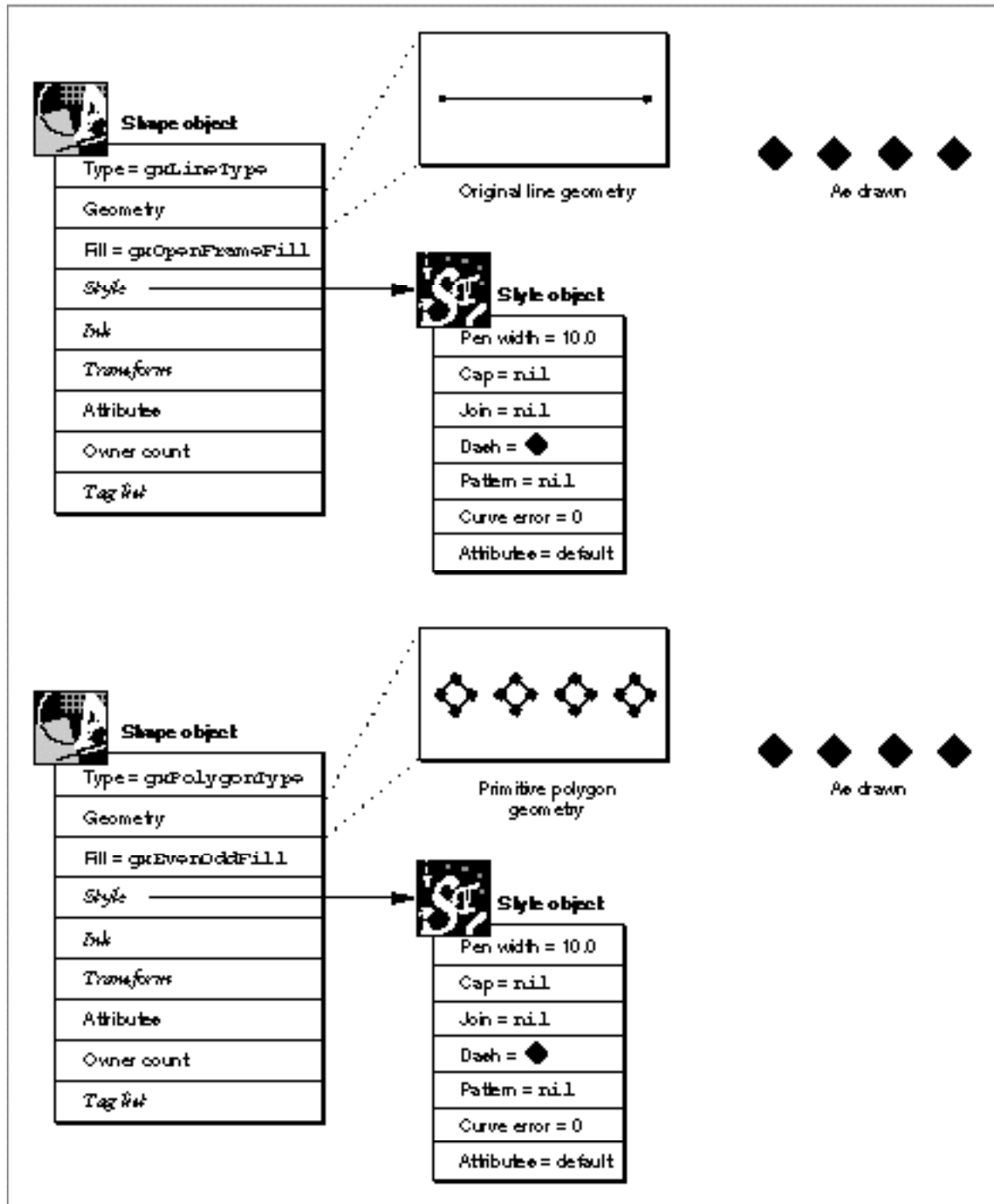
Figure 4-8 shows a simple example of the `GXPrimitiveShape` function. This figure shows a line geometry as drawn with a pen width of 10.0. Converting this line shape to its primitive form results in a rectangle shape with an even-odd fill; the pen width has been incorporated into the geometry of the shape.

**Figure 4-8** Simple example of the GXPrimitiveShape function

Geometric Operations

Figure 4-9 shows a more involved example—a line shape dashed with diamond-shaped polygons. Converting this line shape to its primitive form results in a polygon shape with multiple contours—one contour for each dash.



**Figure 4-9** More involved example of the GXPrimitiveShape function

Notice that, even though the geometry of the shape has changed significantly, the shape appears the same when drawn. Also notice that the `GXPrimitiveShape` function affects only the shape type, shape geometry, and shape fill of a shape—it does not affect the shape’s associated style object. In the example in Figure 4-9, the result of the `GXPrimitiveShape` function has a pen width of 10.0 and dash shape. However, since the shape fill was changed to even-odd fill, these aspects of the style are ignored when the shape is drawn.

For a complete description of the primitive forms of shapes, see the reference description of the `GXPrimitiveShape` function, which is on page 4-79. For some examples that demonstrate when it is necessary to use primitive shapes, see the descriptions of caps, joins, dashes, and patterns in Chapter 3, “Geometric Styles,” in this book, and the description of clip shapes in *Inside Macintosh: QuickDraw GX Objects*.

For programming examples illustrating shapes in their primitive form, see “Converting a Shape to Primitive Form” beginning on page 4-38.

## Geometric Information

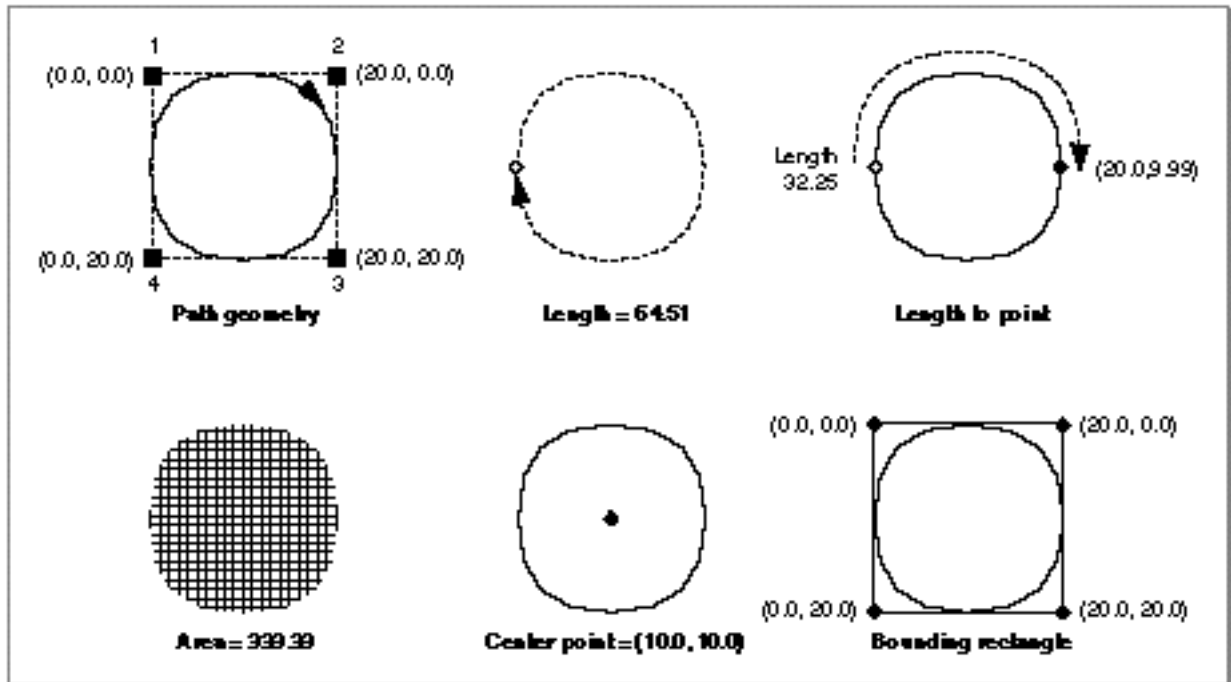
---

QuickDraw GX lets you calculate specific geometric information about a shape, or about the contour of a shape. You can

- n find the length of all of a shape’s contours or of a particular contour of a shape
- n locates the point that falls at a given distance along a particular contour of a shape
- n calculates the area contained by the contours of a shape’s geometry or by a particular contour of a shape’s geometry
- n find the center point of a shape or of a particular contour of a shape
- n find the bounding rectangle of a shape

Figure 4-10 illustrates the geometric information you can obtain about a shape.

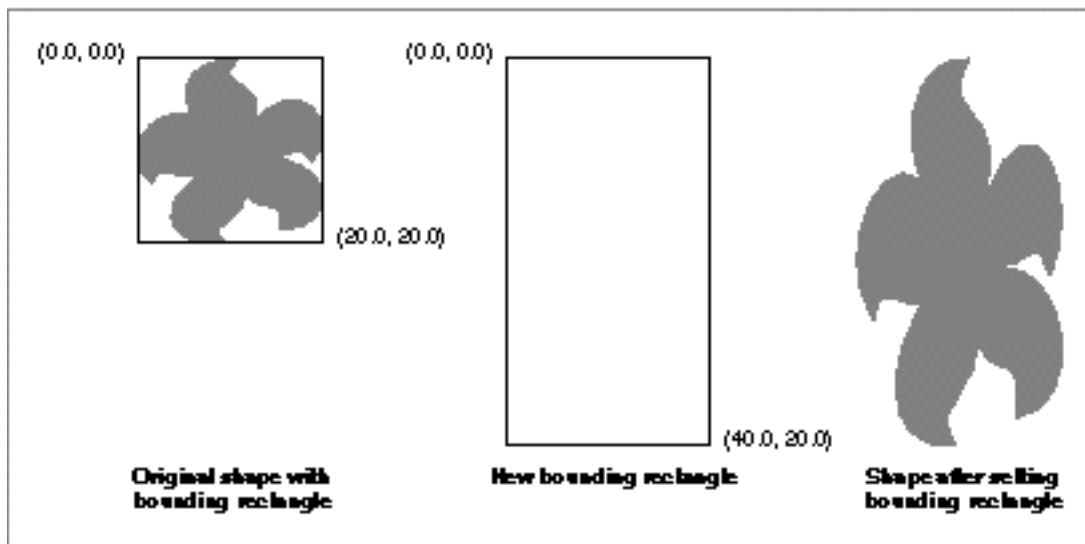
**Figure 4-10** Geometric information available about a path shape



Notice in Figure 4-10 that, because the first point of the path shape is an off-curve control point, the length-to-point operation starts its calculation at an initial on-curve point, halfway between the original first and last points of the contour.

QuickDraw GX also allows you to set the bounding rectangle of a shape, and therefore move and scale the shape, as shown in Figure 4-11.

**Figure 4-11** A path shape resized by changing its bounding rectangle



For programming examples of obtaining geometric information about shapes, see “Finding Geometric Information About a Shape” beginning on page 4-41.

For programming examples of setting the bounding rectangle of a shape, see “Setting a Shape’s Bounding Rectangle” beginning on page 4-47.

## Touching and Containing

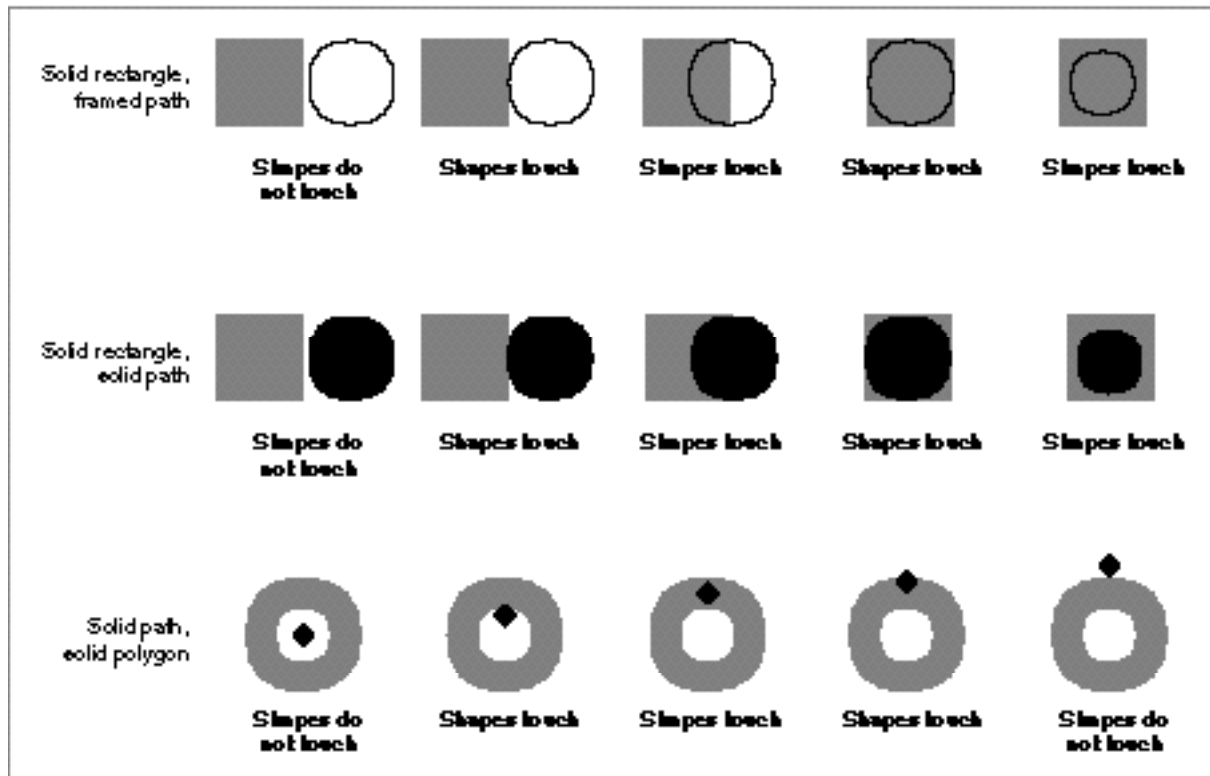
QuickDraw GX allows you to determine if the area enclosed by the contours of one shape touch the area enclosed by the contours of another shape. You can also determine if one shape’s area contains the area of another shape.

In particular, you can

- n determine if a point touches the area enclosed by a rectangle
- n determine if the area enclosed by the contours of a shape touches the area enclosed by a rectangle
- n determine if the areas of two shapes touch

Figure 4-12 shows the results of testing to see whether pairs of different geometric shapes touch. In this figure, a solid rectangle shape is tested for touching with both a framed path and a solid path, and a solid path is tested for touching with a solid polygon.

**Figure 4-12** Testing whether one shape touches another

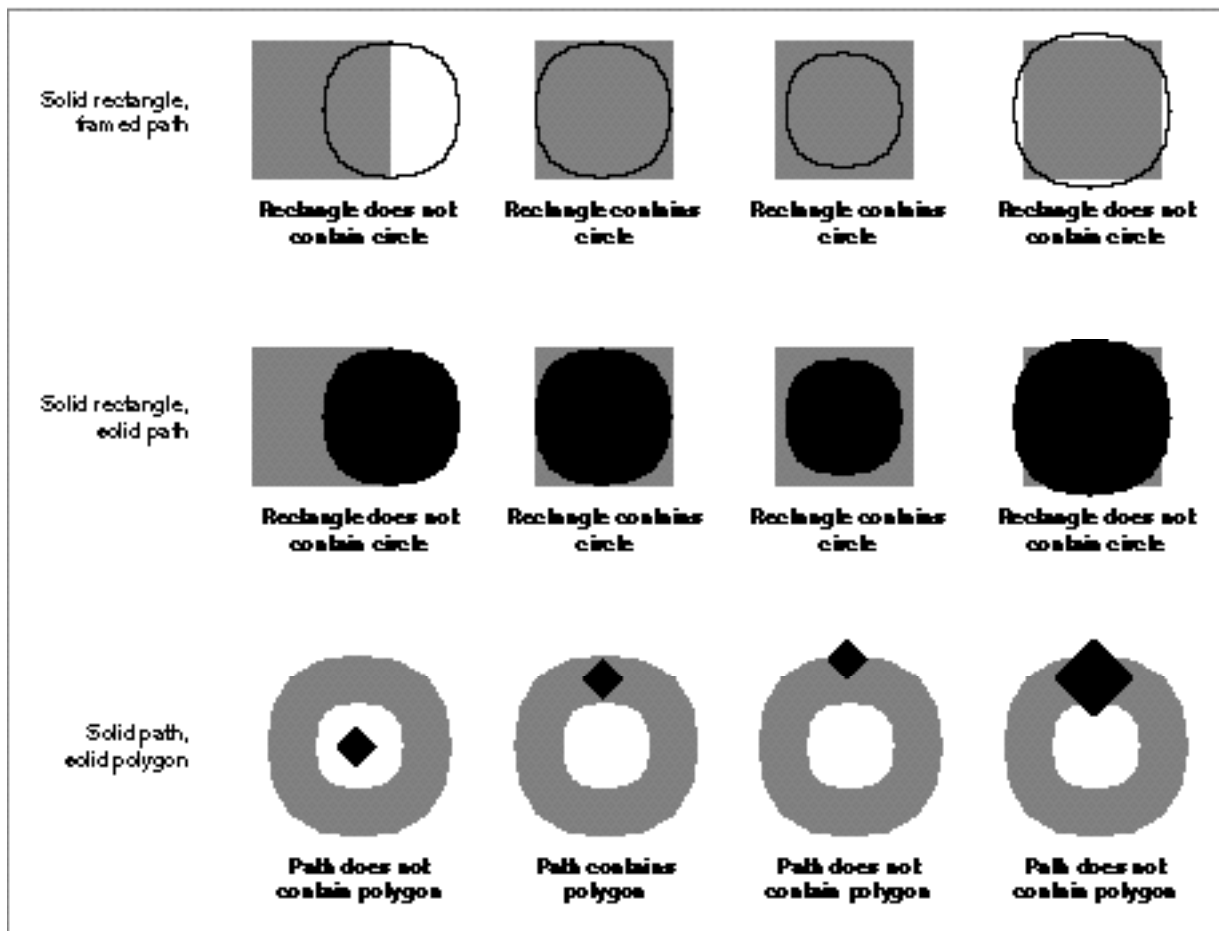


QuickDraw GX also allows you to determine whether or not

- n one rectangle contains another
- n a rectangle contains the area covered by a shape
- n the area covered by one shape contains the area covered by another shape.

Figure 4-13 shows the results of testing pairs of shapes to see if one shape contains another.

**Figure 4-13** Testing whether one shape contains another



Notice the first diagram in the third row of Figure 4-13. A shape does not contain another shape if it merely surrounds the other shape; the area covered by the first shape as drawn must contain the area of the second shape as drawn.

#### Note

QuickDraw GX defines empty shapes as touching no shapes and full shapes as touching any shape except an empty shape. QuickDraw GX also defines full shapes as containing any shape and empty shapes as being contained by any shape except other empty shapes. u

For programming examples of testing shapes for intersection, see “Determining Whether Two Shapes Touch” beginning on page 4-53.

For programming examples of testing shapes for inclusion, see “Determining Whether One Shape Contains Another” beginning on page 4-58.

## Geometric Arithmetic

QuickDraw GX provides six different arithmetic operations that you can perform on geometric shapes. These operations are: intersection, union, difference, reverse difference, exclusion, and inversion. With these operations, you can

- n find the intersection of two rectangles
- n find the union of two rectangles
- n find the area common to two shapes
- n find the combined area of by two shapes
- n find the area covered by one shape that is not also covered by another
- n find the area covered by one shape or another, but not both
- n find the area not covered by a shape

Figure 4-14 illustrates the first five of these arithmetic operations.

**Figure 4-14** Geometric arithmetic with two solid shapes

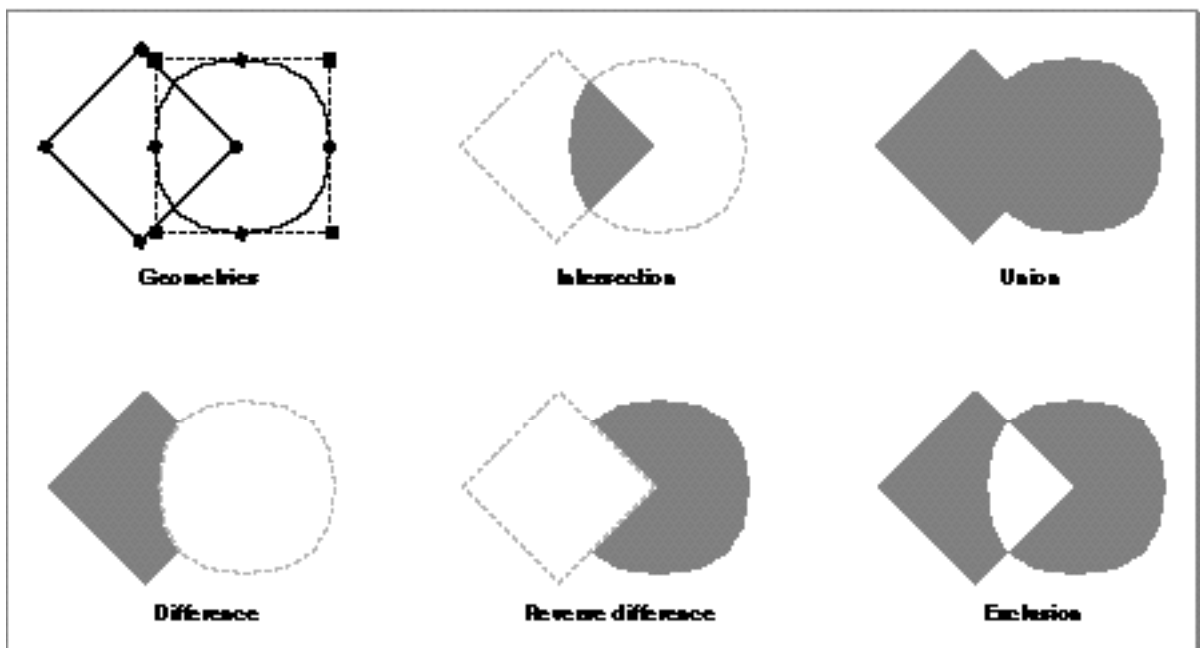


Figure 4-14 shows geometric arithmetic with two solid shapes. You can also perform some geometric arithmetic on a filled shape and a solid shape, as shown in Figure 4-15.

**Figure 4-15** Geometric arithmetic with a framed shape and a solid shape

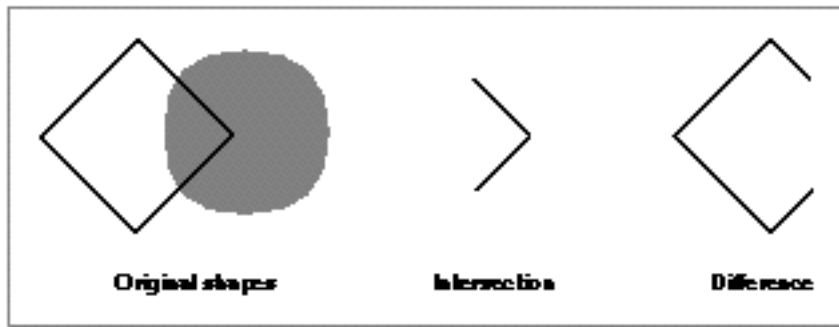
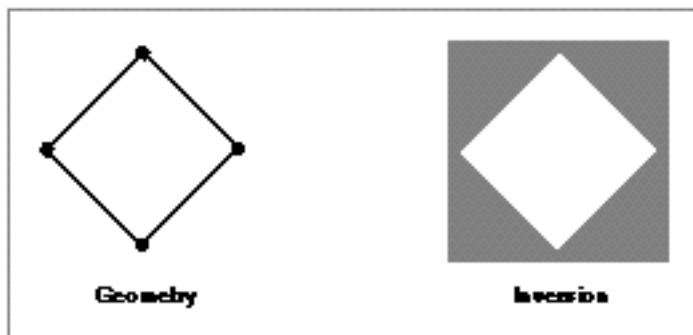


Figure 4-16 illustrates the geometric inversion—the area not covered by a shape. The inverted shape extends to the limits of its clip shape or the limits of the view port to which it is drawn.

**Figure 4-16** Geometric inversion



For programming examples of geometric arithmetic, see “Performing Geometric Arithmetic With Shapes” beginning on page 4-60.



## Using Geometric Operations

---

This section shows you how to apply geometric operations to shapes. In particular, this section shows you how to

- n determine and reverse the contour direction of a shape's contours
- n break a contour into multiple contours
- n reduce and simplify the geometric description of a shape
- n incorporate style information into a shape's geometry
- n obtain geometric information about a shape's geometry, such as contour length and area
- n determine and alter the bounding rectangle of a shape
- n inset a shape's geometry
- n determine if two shapes touch
- n determine if one shape contains another
- n perform geometric arithmetic, such as intersection and union, on shapes

Many of the sample functions in this section create geometric shapes, and to do so, they specify geometric points for the shapes' geometries. Since a geometric point contains two fixed-point values, the sample functions in this section must convert integer constants to fixed-point constants when specifying a geometric point. QuickDraw GX provides the `GXIntToFixed` macro, which performs this conversion by shifting the integer value 16 bits to the left:

```
#define GXIntToFixed(a) ((Fixed) (a) << 16)
```

QuickDraw GX also provides the `ff` macro as a convenient alias:

```
#define ff(a) GXIntToFixed(a)
```

The `ff` macro is used throughout this section.

### Determining and Reversing Contour Direction

---

The contours of geometric shapes have contour direction: either clockwise or counterclockwise, as described in "Contours and Contour Direction" beginning on page 4-4. QuickDraw GX allows you to determine the contour direction of a specific contour of a shape and also allows you to change the direction of a shape's contour by reversing the order of the geometric points in the geometry defining the contour.

The sample function in Listing 4-1 creates a polygon shape with two contours—one having a clockwise contour direction and the other having a counterclockwise contour direction.

---

**Listing 4-1**     Creating a polygon shape with two contours having opposite contour directions

```
void CreateConcentricTriangles(void)
{
    gxShape twoTriangles;

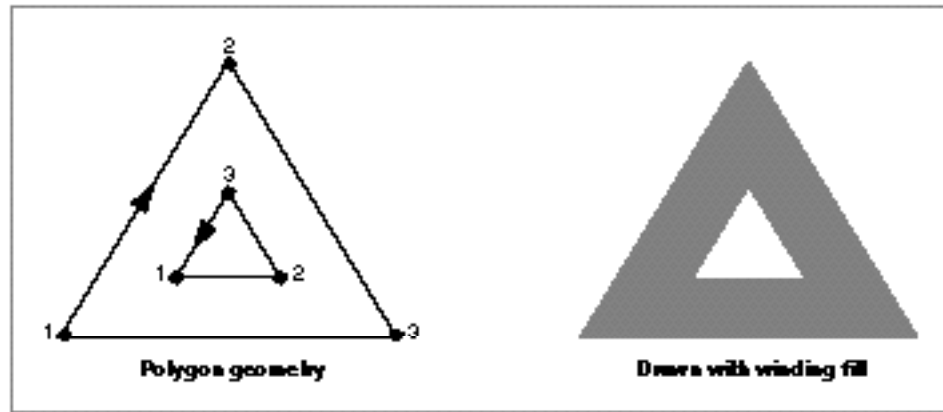
    long twoTrianglesGeometry[] = {2, /* number of contours */
                                    3, /* number of points */
                                    ff(50), ff(200),
                                    ff(110), ff(100),
                                    ff(170), ff(200),
                                    3, /* number of points */
                                    ff(90), ff(178),
                                    ff(130), ff(178),
                                    ff(110), ff(145)};

    twoTriangles = GXNewPolygons((gxPolygons *)
                                twoTrianglesGeometry);
    GXSetShapeFill(twoTriangles, gxWindingFill);

    GXDrawShape(twoTriangles);
    GXDisposeShape(twoTriangles);
}
```

The result of this sample function is shown in Figure 4-17.

**Figure 4-17** A polygon shape whose two contours have opposite contour directions



QuickDraw GX provides the `GXGetShapeDirection` function to allow you to determine the contour direction of a specific contour in a shape. This function takes two parameters: the first parameter is a reference to the shape and the second parameter is the index of the contour whose contour direction you want to find. In the example from Listing 4-1, the first contour (the outer contour) has a clockwise contour direction. Calling the function

```
GXGetShapeDirection(twoTriangles, 1);
```

returns the constant `gxClockwiseDirection`.

The second contour (the inner contour) has a counterclockwise direction. Calling the function

```
GXGetShapeDirection(twoTriangles, 2);
```

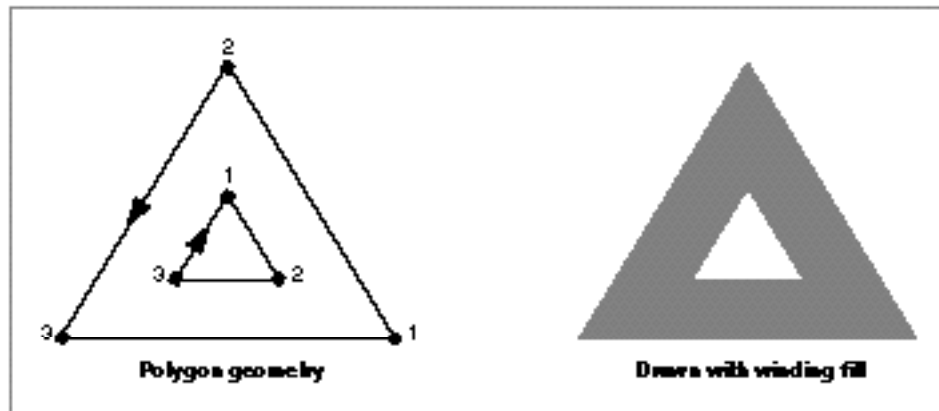
returns the constant `gxCounterclockwiseDirection`.

You can reverse the direction of a contour by reversing the order of the contour's geometric points. For this purpose, QuickDraw GX provides the `GXReverseShape` function. This function also takes two parameters: a reference to the shape and the index of the contour to reverse. Specifying 0 as the number of the contour to reverse causes the `GXReverseShape` function to reverse all the contours of a shape. For example, you can add the following function call to the sample function in Listing 4-1:

```
GXReverseShape(twoTriangles, 0);
```

The result is shown in Figure 4-18.

**Figure 4-18** A polygon shape with the direction of both contours reversed



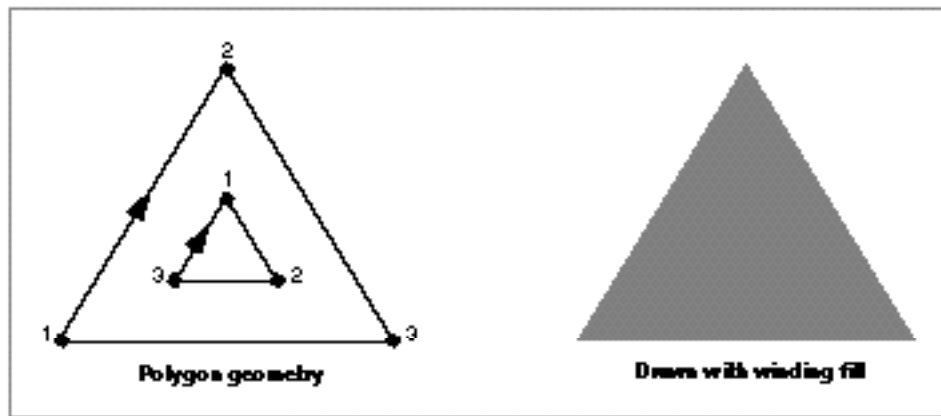
Since both contours are reversed in this example, the shape appears the same when drawn as it did before the contours were reversed.

However, reversing only the inner contour of this polygon by calling

```
GXReverseShape(twoTriangles, 2);
```

results in the polygon shown in Figure 4-19.

**Figure 4-19** A polygon shape with the direction of the inner contour reversed



Reversing the contour of a shape by calling the `GXReverseShape` function almost always changes the result of the `GXGetShapeDirection` function. One important exception, however, is that line shapes always have a clockwise direction. The order of a line shape's geometric points does not affect the result of the `GXGetShapeDirection` function.

For a discussion of contour direction, see “Contours and Contour Direction” beginning on page 4-4.

For more information about the `GXGetShapeDirection` function, see page 4-68. For more information about the `GXReverseShape` function, see page 4-70.

## Breaking Shape Contours

---

Polygon and path shapes can contain many contours. Each contour of a polygon shape can be made up of many lines and each contour of a path shape can be made up of many lines and curves.

QuickDraw GX provides a method for breaking a single contour of a polygon or path shape into two contours at a specified geometric point in the original contour.

As an example, the sample function in Listing 4-2 creates a path shape with a single contour. This contour contains six geometric points and is made up of a curve, a line, and another curve.

---

**Listing 4-2**      Creating a path shape with a single contour

```
void CreateSingleContourPath(void)
{
    gxShape  aPathShape;

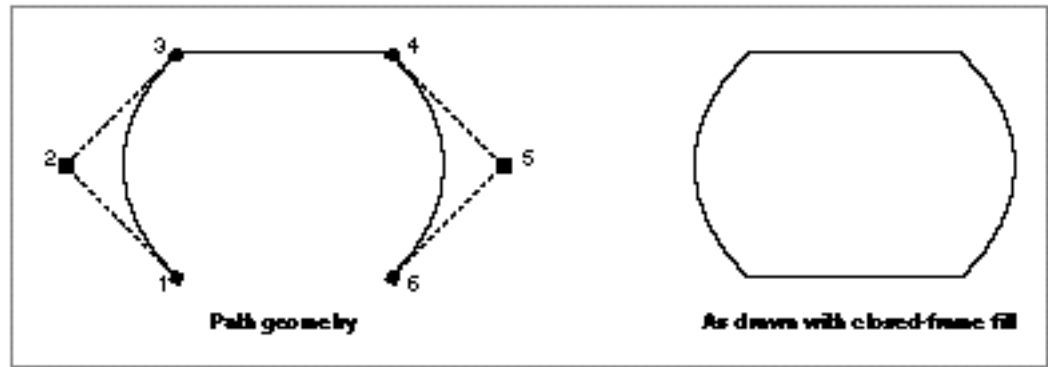
    static long oneContourGeometry[] = {1, /* number of contours */
                                         6, /* number of points */
                                         0x48000000, /* 0100 1000 */
                                         ff(100), ff(150), /* on */
                                         ff(50), ff(100), /* off */
                                         ff(100), ff(50), /* on */
                                         ff(200), ff(50), /* on */
                                         ff(250), ff(100), /* off */
                                         ff(200), ff(150)}; /* on */

    aPathShape = GXNewPaths((gxPaths *) oneContourGeometry);
    GXSetShapeFill(aPathShape, gxClosedFrameFill);

    GXDrawShape(aPathShape);
    GXDisposeShape(aPathShape);
}
```

The result of this function is shown in Figure 4-20.

**Figure 4-20** A path shape with a single contour

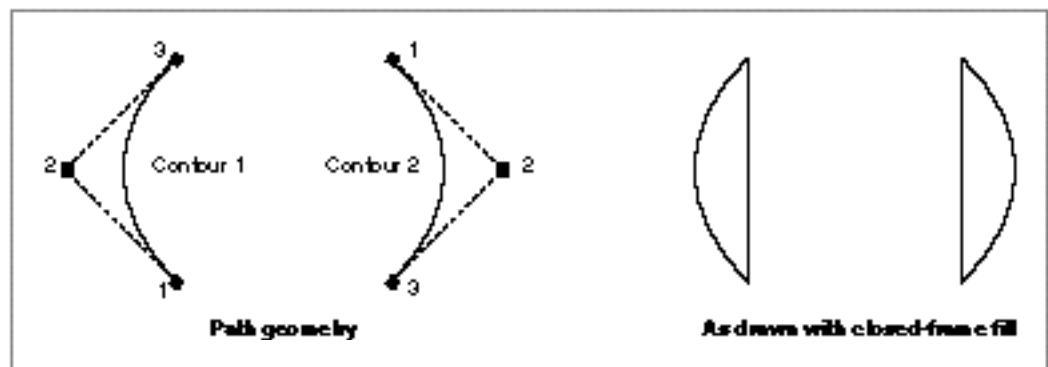


The `GXBreakShape` function allows you to break a single contour into two contours at a specified geometric point. Adding the function call

```
GXBreakShape(aPathShape, 4);
```

to the sample function in Listing 4-2 breaks the single contour of the path shape into two contours at the fourth geometric point, as shown in Figure 4-21.

**Figure 4-21** A path shape broken into two contours



## Geometric Operations

After the call to the `GXBreakShape` function, the path shape has two contours, each with three geometric points. Calling the function

```
GXCountShapeContours(aPathShape);
```

returns the value 2.

In addition to breaking the contours of polygon and path shapes, you can also use the `GXBreakShape` function to break line shapes and curve shapes. For example, if the variable `aLine` references a line shape, the function call

```
GXBreakShape(aLine, 1);
```

converts the line shape to a polygon shape with two contours. The first contour is empty (that is, it has no geometric points) and the second contour is the original line. Calling the function

```
GXCountShapeContours(aLine);
```

returns the value 2.

For a discussion of contours, geometric points, and the `GXCountShapeContours` function, see Chapter 2, “Geometric Shapes,” in this book.

You can also use the `GXSetPolygonParts`, `GXSetPathParts`, and `GXSetShapeParts` functions to break a shape’s contours. These functions are also described in Chapter 2, “Geometric Shapes,” in this book.

For more information about the `GXBreakShape` function, see page 4-72.

## Eliminating Unnecessary Geometric Points

---

There are many ways in which polygon and path shapes can contain more geometric points than necessary to describe their underlying geometry. Two common examples are

- n duplicate points—sequential points with the same coordinates
- n colinear points—points that lie in a straight line between a preceding point and subsequent point



The sample function in Listing 4-3 creates a polygon shape with a single contour that has six geometric points, two of which are unnecessary.

**Listing 4-3** Creating a polygon with redundant geometric points

```
void ReduceUnnecessaryPoints(void)
{
    gxShape squareShape;

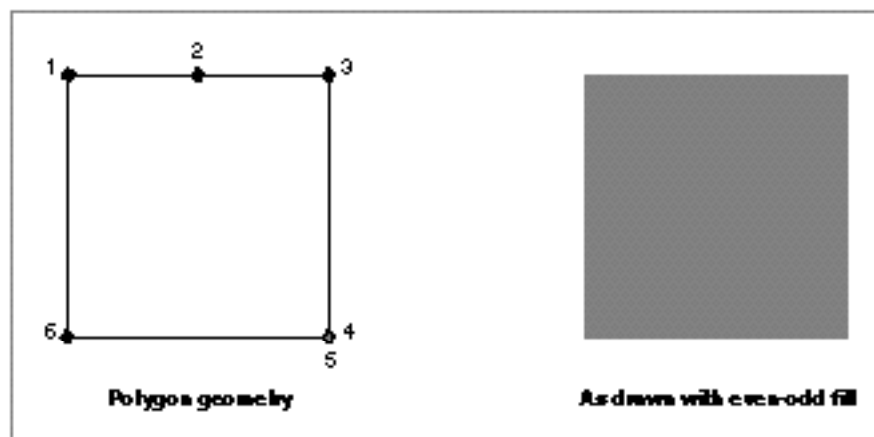
    static long paddedSquareGeometry[] = {1, /* # of contours */
                                           6, /* # of contours */
                                           ff(50), ff(50),
                                           ff(100), ff(50),
                                           ff(150), ff(50),
                                           ff(150), ff(150),
                                           ff(150), ff(150),
                                           ff(50), ff(150)};

    squareShape = GXNewPolygons((gxPolygons *)
                                &paddedSquareGeometry);
    GXSetShapeFill(squareShape, gxEvenOddFill);

    GXDrawShape(squareShape);
    GXDisposeShape(squareShape);
}
```

The resulting polygon shape is shown in Figure 4-22.

**Figure 4-22** A polygon shape with unnecessary geometric points



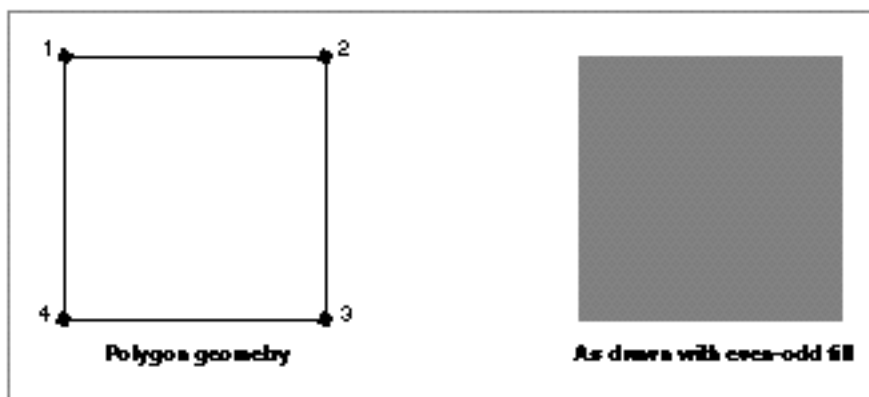
QuickDraw GX provides the `GXReduceShape` function so you can eliminate unnecessary duplicate and colinear points. The `GXReduceShape` function takes two parameters: a reference to the shape containing the contour whose unnecessary geometric points you want to eliminate and an index specifying the contour itself. If you supply the value 0 for the second parameter, the `GXReduceShape` function eliminates unnecessary geometric points from all the contours of a shape.

As an example, adding the function call

```
GXReduceShape(squareShape, 0);
```

to the sample function in Listing 4-3 results in the polygon shape shown in Figure 4-23.

**Figure 4-23** A polygon shape with the unnecessary geometric points removed



The unnecessary duplicate geometric point and the unnecessary colinear geometric point are gone, but the polygon still appears the same when drawn. Although the resulting geometry could be described by a rectangle shape, the shape in this example remains a polygon shape. The `GXReduceShape` function does not convert the shape type of the original shape. (However, the `GXSimplifyShape` function, shown in the next section, does convert shape type, when possible.)

The `GXReduceShape` function considers two points to be duplicate points if they are within the distance from each other specified by the curve error property of the shape's style object. See Chapter 3, "Geometric Styles," in this book for a discussion of curve error.

For a discussion of geometric points, see Chapter 2, "Geometric Shapes," in this book.

For more information about the `GXReduceShape` function, see page 4-74.

## Simplifying Shapes

---

In addition to unnecessary geometric points, there are other aspects of shape geometries that complicate the definition and drawing of a shape. Some examples are:

- n an unnecessary contour break, where an open-framed contour ends on the same point where the subsequent contour begins
- n a crossed contour, where a contour crosses over itself or another contour of the same shape
- n overlapping contours, where inner contour loops have the same contour direction as the contour that contains them

The sample function in Listing 4-4 creates a polygon shape with a single contour that crosses over itself.

---

**Listing 4-4**      Creating a polygon shape with a crossed contour

```
void CreateHourglassPolygon(void)
{
    gxShape aPolygonShape;

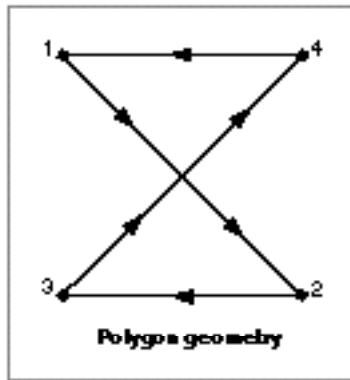
    static long hourglassGeometry[] = {1, /* number of contours */
                                        4, /* number of points */
                                        ff(50), ff(50),
                                        ff(150), ff(50),
                                        ff(50), ff(150),
                                        ff(150), ff(150)};

    aPolygonShape = GXNewPolygons((gxPolygons *)
                                   hourglassGeometry);
    GXSetShapeFill(aPolygonShape, gxClosedFrameFill);

    GXDrawShape(aPolygonShape);
    GXDisposeShape(aPolygonShape);
}
```

The resulting polygon shape is shown in Figure 4-24.

**Figure 4-24** A polygon shape with a crossed contour



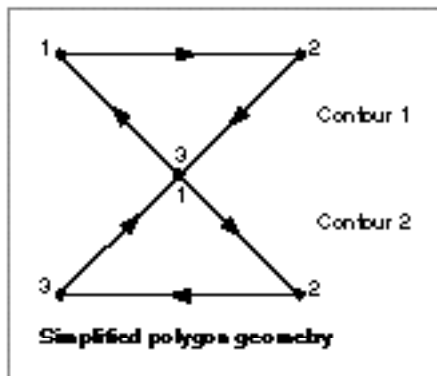
QuickDraw GX provides the `GXSimplifyShape` function so you can eliminate unnecessary contour breaks, crossed contours, and overlapping contours. This function takes one parameter: a reference to the shape you want to simplify.

As an example, adding the function call

```
GXSimplifyShape(aPolygonShape);
```

to the sample function in Listing 4-4 creates the polygon shown in Figure 4-25.

**Figure 4-25** A polygon shape with no crossed contours



Notice that although this polygon shape is simplified, it contains more geometric points and more contours than the original polygon. However, the crossed contour is eliminated.

As another example, the sample function in Listing 4-5 creates a path shape with two concentric contours: an outer contour and an inner contour, both of which have a clockwise contour direction.

---

**Listing 4-5**      Creating a path shape with two clockwise contours

```
void CreateConcentricPaths(void)
{
    gxShape aPathShape;

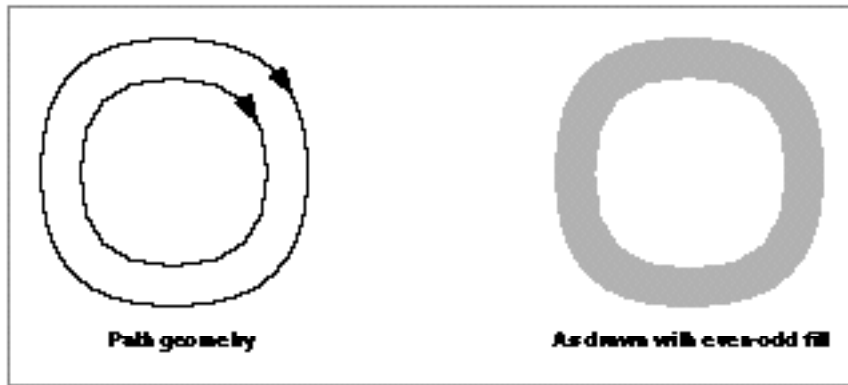
    static long twoCircleGeometry[] = {2, /* # of contours */
                                        4, /* # of points */
                                        0xF0000000, /* 1111 ... */
                                        ff(50), ff(50), /* off */
                                        ff(150), ff(50), /* off */
                                        ff(150), ff(150), /* off */
                                        ff(50), ff(150), /* off */
                                        4, /* # of points */
                                        0xF0000000, /* 1111 ... */
                                        ff(65), ff(65), /* off */
                                        ff(135), ff(65), /* off */
                                        ff(135), ff(135), /* off */
                                        ff(65), ff(135)}; /* off */

    aPathShape = GXNewPaths((gxPaths *) twoCircleGeometry);
    GXSetShapeFill(aPathShape, gxEvenOddFill);

    GXDrawShape(aPathShape);
    GXDisposeShape(aPathShape);
}
```

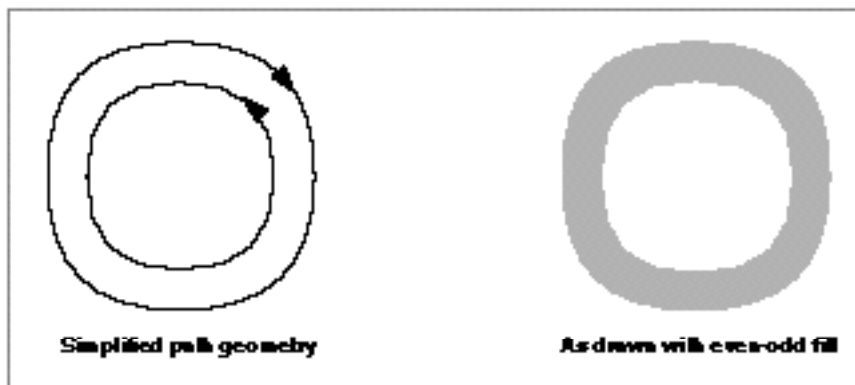
Figure 4-26 shows the result of this sample function.

**Figure 4-26** A path shape with two concentric clockwise contours and even-odd shape fill



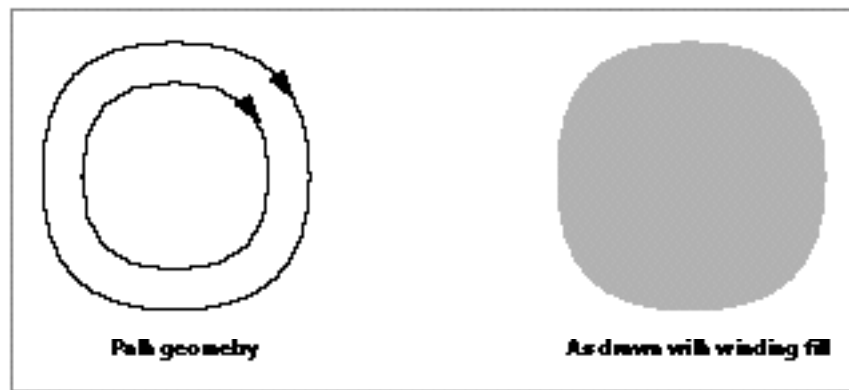
Applying the `GXSimplifyShape` function to the path shape in Figure 4-26 reverses the contour direction of the inner contour, so that it is no longer an overlapping contour with the same contour direction. The result is shown in Figure 4-27.

**Figure 4-27** A path shape with two concentric contours with opposite contour direction



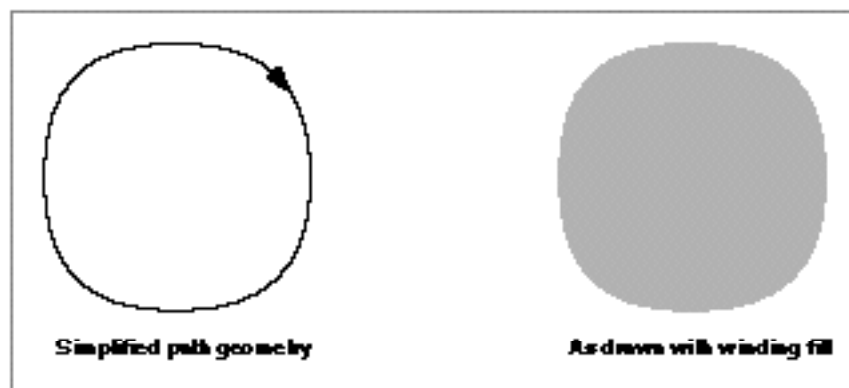
However, imagine that the path shape defined in Listing 4-5 originally had a winding fill, as shown in Figure 4-28.

**Figure 4-28** A path shape with two concentric clockwise contours and winding shape fill



In this case, the `GXSimplifyShape` function removes the inner contour entirely, as it is not necessary to describe the shape as drawn. The result is shown in Figure 4-29.

**Figure 4-29** A path shape simplified to a single clockwise contour



The `GXSimplifyShape` function can change the shape type of a shape, as well the geometry of shape, if the shape can be expressed by a simpler shape type. For example, a polygon shape is converted to a rectangle shape or a line shape, if possible. Similarly, a path shape is converted to a polygon shape if it has no off-curve control points.

For a discussion of shape fills and contour direction, see Chapter 2, “Geometric Shapes,” in this book.

For more information about the `GXSimplifyShape` function, see page 4-76.

## Converting a Shape to Primitive Form

---

QuickDraw GX requires that certain shapes (such as cap shapes, join shapes, dash shapes, pattern shapes, and clip shapes) be in primitive form—that is, they must have all of their style modifications incorporated into their geometries. Before you set a cap shape, join shape, dash shape, and so on, you must ensure that the shape is in primitive form.

As an example of converting a shape to primitive form, the sample function in Listing 4-6 creates a polygon shape that is not in primitive form. The polygon has one contour, a closed-frame shape fill, and a pen width of 15.0.

---

**Listing 4-6**      Creating an hourglass polygon shape with a thick pen width

```
void CreateHourglassPolygon(void)
{
    gxShape aPolygonShape;
    gxJoinRecord theJoinRecord;

    static long hourglassGeometry[] = {1, /* number of contours */
                                        4, /* number of points */
                                        ff(50), ff(50),
                                        ff(150), ff(50),
                                        ff(50), ff(150),
                                        ff(150), ff(150)};

    aPolygonShape = GXNewPolygons((gxPolygons *)
                                  hourglassGeometry);
    GXSetShapeFill(aPolygonShape, gxClosedFrameFill);
    GXSetShapePen(aPolygonShape, ff(15));

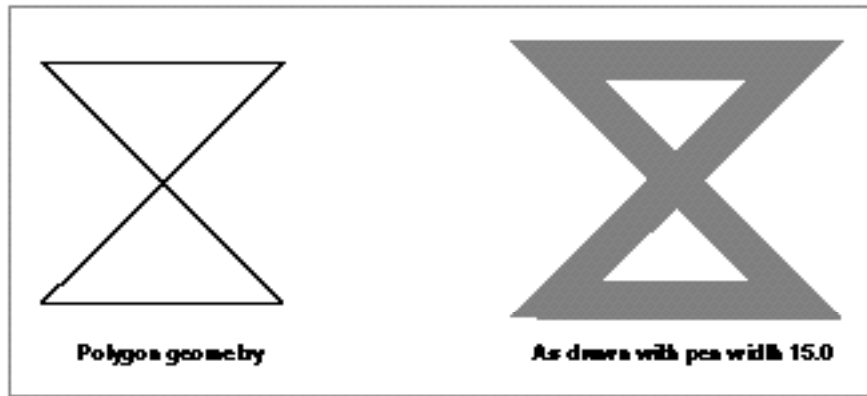
    theJoinRecord.attributes = gxSharpJoin;
    theJoinRecord.join = nil;
    theJoinRecord.miter = gxPositiveInfinity;
    GXSetShapeJoin(aPolygonShape, &theJoinRecord);

    GXDrawShape(aPolygonShape);
    GXDisposeShape(aPolygonShape);
}
```



The result of this sample function is shown in Figure 4-30.

**Figure 4-30** A hourglass-shaped polygon with a thick border



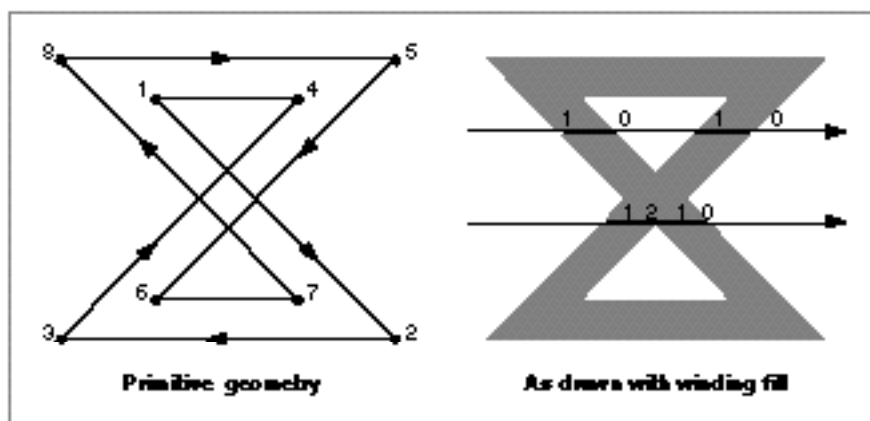
The polygon shape defined in Listing 4-6 is not in primitive form because an element of the polygon's style (the pen width) is not incorporated into the polygon's geometry.

QuickDraw GX provides the `GXPrimitiveShape` function so you can incorporate the elements of a shape's style into the shape's geometry. For example, adding the function call

```
GXPrimitiveShape(aPolygonShape);
```

to the sample function in Listing 4-6 creates the polygon shown in Figure 4-31.

**Figure 4-31** A polygon shape with style information incorporated into its geometry



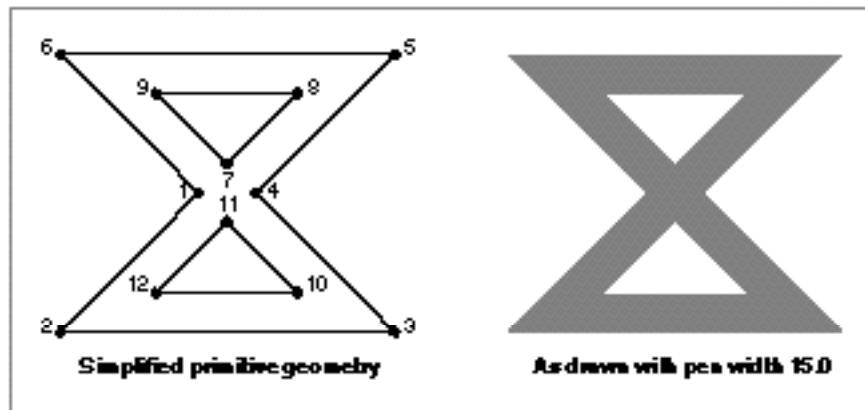
As shown in Figure 4-31, the `GXPrimitiveShape` function has incorporated the pen width into the geometry of the polygon; the resulting polygon has two contours whereas the original had one, and the resulting polygon has a winding fill instead of a closed-frame fill.

Notice that the primitive form of this polygon is not simplified because the `GXPrimitiveShape` function does not simplify its result. You can simplify the result of this function by calling

```
GXSimplifyShape(aPolygonShape);
```

Figure 4-32 shows the resulting shape—the primitive form of the polygon with no crossed or overlapping contours.

**Figure 4-32** The primitive form of the polygon shape after simplification



The polygon shape now has three contours—which do not cross or overlap—and yet it appears the same as the original polygon shape when drawn.

For a discussion of style modifications and more examples of the `GXPrimitiveShape` function, see Chapter 3, “Geometric Styles,” in this book.

For more information about the `GXPrimitiveShape` function, see page 4-79.

## Finding Geometric Information About a Shape

---

QuickDraw GX provides a number of functions that allow you to determine geometric information about a shape, such as the length of a contour or the area covered by a shape.

The sample function in Listing 4-7 creates a path shape with two concentric contours, the outer contour having a clockwise contour direction and the inner contour having a counterclockwise contour direction. This path shape is used in subsequent sections to illustrate the geometric information functions.

---

**Listing 4-7**      Creating a path shape with two contours having opposite contour directions

```
void CreateConcentricCircles(void)
{
    gxShape aPathShape;

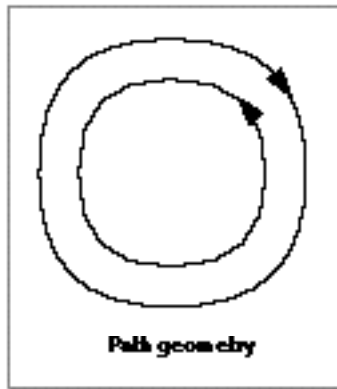
    static long twoCircleGeometry[] = {2, /* number of contours */
                                        4, /* number of points */
                                        0xF0000000, /* 1111 ... */
                                        ff(50), ff(50), /* off */
                                        ff(150), ff(50), /* off */
                                        ff(150), ff(150), /* off */
                                        ff(50), ff(150), /* off */
                                        4, /* number of points */
                                        0xF0000000, /* 1111 ... */
                                        ff(65), ff(135), /* off */
                                        ff(135), ff(135), /* off */
                                        ff(135), ff(65), /* off */
                                        ff(65), ff(65)}; /* off */

    aPathShape = GXNewPaths((gxPaths *) twoCircleGeometry);

    GXDrawShape(aPathShape);
    GXDisposeShape(aPathShape);
}
```

The resulting shape geometry is shown in Figure 4-33.

**Figure 4-33** A path with an outer clockwise contour and an inner counterclockwise contour



### Finding the Length of a Contour

QuickDraw GX provides the `GXGetShapeLength` function so you can measure the length of a contour. This function takes three parameters: a reference to the shape containing the contour you want to measure, an index indicating which contour you want to measure, and a pointer to a variable of type `gxWide` to store the result.

For example, if you add the declaration

```
gxWide length;
```

and the function call

```
GXGetShapeLength(aPathShape, 1, &length);
```

to the sample function in Listing 4-7, the value returned in the `length` parameter is approximately 322.543, which is the length (the circumference) of the outer contour.

For more information about the `GXGetShapeLength` function, see page 4-83.

### Finding the Point at a Certain Distance Along a Contour

QuickDraw GX provides the `GXShapeLengthToPoint` function that allows you to calculate the position of the point that falls at a specified distance along a contour. This function also calculates the tangent of the contour at that point.

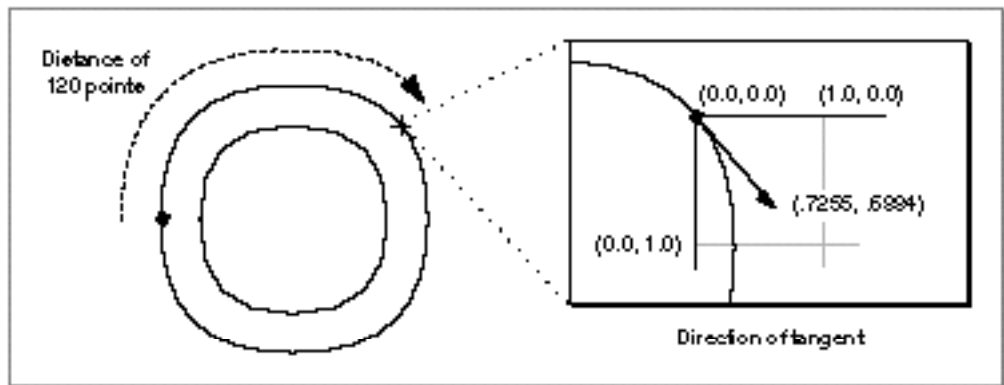
As an example, adding the function call

```
GXShapeLengthToPoint(aPathShape, 1, ff(120),
                    &thePoint, &theDirection);
```

to the sample function in Listing 4-7 determines the point that falls along the first contour at a distance of 120.0 points from the start of the contour, and stores the resulting point in the `thePoint` parameter. Also, in the `theDirection` parameter, this function stores a tangent vector indicating the direction of the contour at that point.

The result of this function is shown in Figure 4-34.

**Figure 4-34** Finding a specified point on a path contour



For more information about the `GXShapeLengthToPoint` function, see page 4-85.

### Finding the Bounding Rectangle and Center Point of a Shape

QuickDraw GX provides functions for finding the bounding rectangle of a shape and the center point of a shape. The bounding rectangle is the smallest rectangle that contains the shape. The center point of a shape is not the center of the shape's bounding rectangle; rather it is the "center of gravity" of a shape. QuickDraw GX guarantees that the center point of a shape remains the same even if the shape is rotated.

You can use the `GXGetShapeBounds` function to find the bounding rectangle of a shape. As an example, if you apply the function

```
GXGetShapeBounds(aPathShape, 0, &theBounds);
```

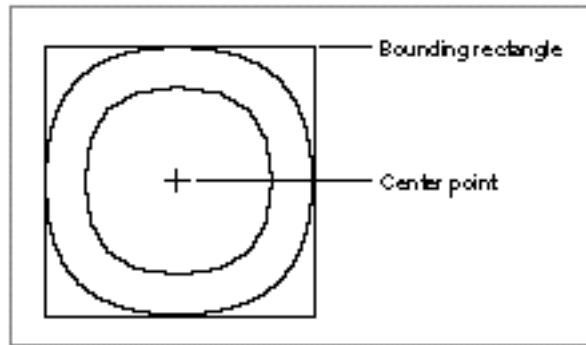
to the path shape from Listing 4-7, the result is a rectangle with the coordinates (50.0, 50.0, 150.0, 150.0). Similarly, if you apply the function

```
GXGetShapeCenter(aPathShape, 0, &thePoint);
```

to the same path shape, the result is the point: (100.0, 100.0).

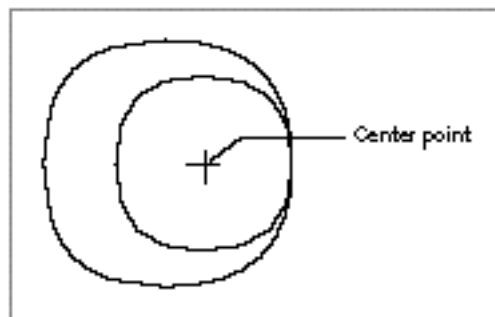
The results of these functions are depicted in Figure 4-35.

**Figure 4-35** Finding the bounding rectangle and the center point of a path



If you move the inner contour of the path shape to right, the center point moves to the right as well, effectively moving with the combined “center of gravity” of the two contours, as shown in Figure 4-36.

**Figure 4-36** Finding the center point of two contours



Notice that the center point lies somewhere between the center of the outer contour and the center of the inner contour.

For more information about the `GXGetShapeBounds` function, see page 4-90. For more information about the `GXGetShapeCenter` function, see page 4-87.

## Finding the Area of a Shape

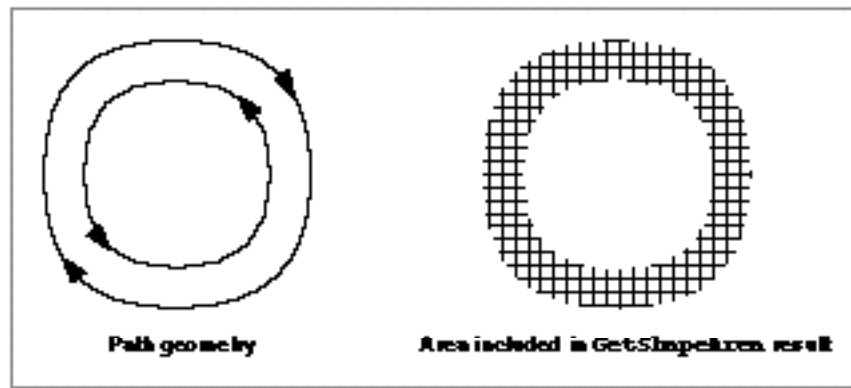
QuickDraw GX provides the `GXGetShapeArea` function so you can determine the area covered by a shape.

With the path shape from Listing 4-7 (on page 4-41), applying the function

```
GXGetShapeArea(aPathShape, 1, &theArea);
```

results in the value 4250.0. This value represents the area of the outer contour minus the area of the inner contour, as shown in Figure 4-37.

**Figure 4-37** Finding the area of a path, two contours with same contour direction



In effect, the function finds the area covered by the shape as if it were filled with the winding shape fill.

Therefore, if you reverse the direction of the inner contour of this path with the function call

```
GXReverseShape(aPathShape, 2);
```

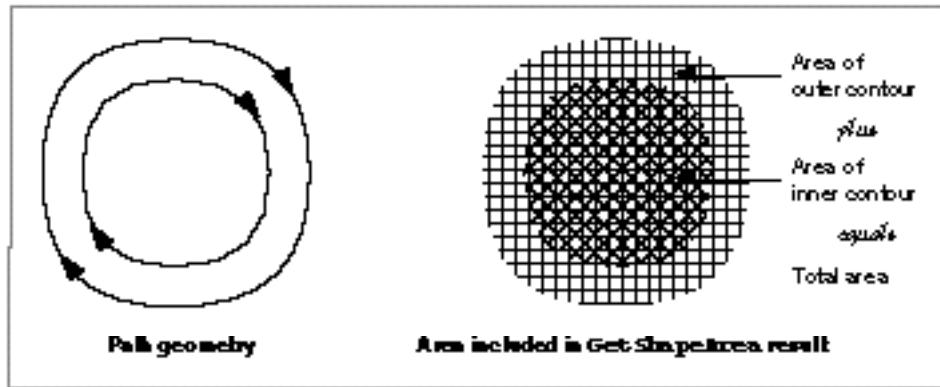
then the function call

```
GXGetShapeArea(aPathShape, 1, &theArea);
```

results in the value 12416.6666. This value represents the area of the outer contour *plus* the area of the inner contour—the area covered by the inner contour is counted twice.

The area included in this calculation is depicted in Figure 4-38.

**Figure 4-38** Finding the area of a path, two contours with opposite contour direction



Note that the `GXGetShapeArea` function does not consider the shape fill when calculating area—it includes this overlapping area twice whether the shape fill is winding fill, even-odd fill, open-frame fill, or closed-frame fill.

You can correct this calculation by calling the `GXSimplifyShape` function first. For example, if you set the shape fill to winding fill with the function call

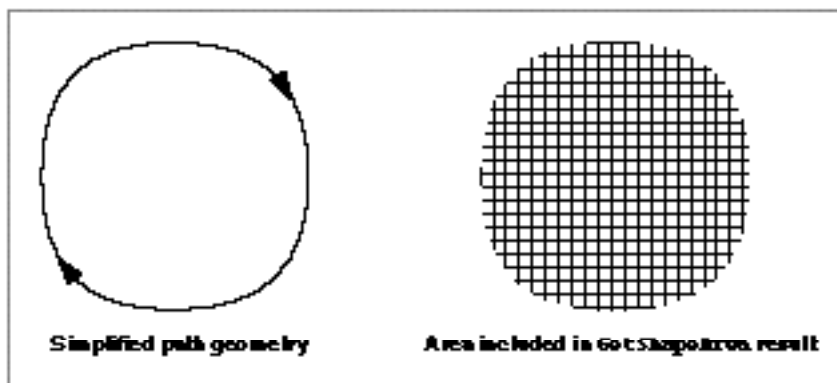
```
GXSetShapeFill(aPathShape, gxWindingFill);
```

and then call the `GXSimplifyShape` function:

```
GXSimplifyShape(aPathShape);
```

the `GXSimplifyShape` function removes the inner contour, as shown in Figure 4-39.

**Figure 4-39** Finding the area of a simplified path





Once this inner contour is removed, you can call the `GXGetShapeArea` function, and the area of the original outer contour (8333.3333) is returned.

For more information about the `GXGetShapeArea` function, see page 4-88.

## Setting a Shape's Bounding Rectangle

---

The `GXGetShapeBounds` function, illustrated on page 4-44, allows you to determine the bounding rectangle of a shape. Similarly, the `GXSetShapeBounds` function allows you to alter the bounding rectangle of a shape, thereby scaling the shape to a new size and moving it to a new location.

As an example, the sample function in Listing 4-8 creates a path with a single circular contour.

---

**Listing 4-8**      Creating a circular path

```
void CreateCircularPath(void)
{
    gxShape      aPathShape;

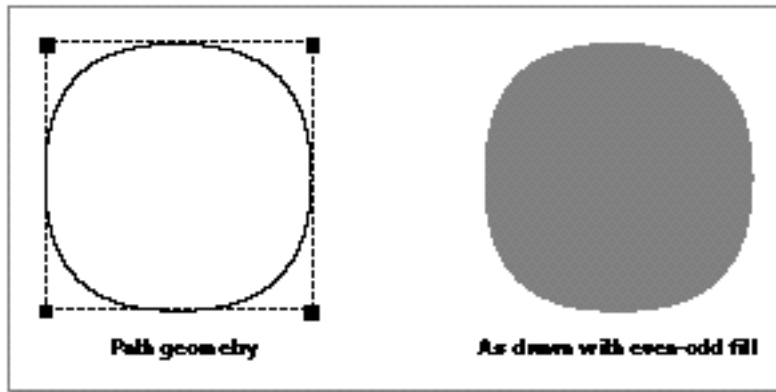
    static long  circularGeometry[] = {1, /* # of contours */
                                        4, /* # of points */
                                        0xF0000000, /* 1111 ... */
                                        ff(50), ff(50), /* off */
                                        ff(150), ff(50), /* off */
                                        ff(150), ff(150), /* off */
                                        ff(50), ff(150)}; /* off */

    aPathShape = GXNewPaths((gxPaths *) circularGeometry);

    GXDrawShape(aPathShape);
    GXDisposeShape(aPathShape);
}
```

The result of this function is shown in Figure 4-40.

**Figure 4-40** A circular path



The bounding rectangle of this shape, which you can determine by calling

```
GXGetShapeBounds(aPathShape, 0, &theBounds);
```

is (50.0, 50.0, 150.0, 150.0). You can move and resize this shape by declaring a new bounding rectangle

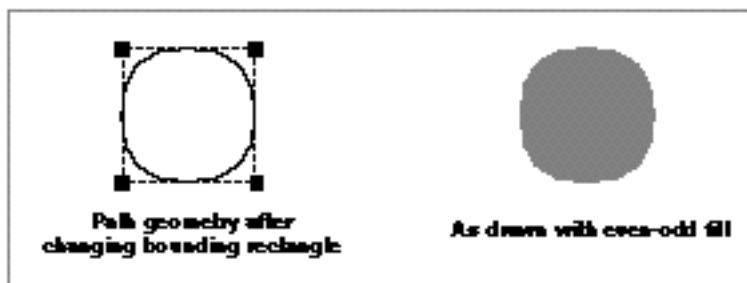
```
gxRectangle newBounds = {ff(60), ff(60), ff(110), ff(110)};
```

and then calling the function

```
GXSetShapeBounds(aPathShape, &newBounds);
```

The geometry of the altered shape is centered around the point (85.0, 85.0) and is smaller than the original shape, as shown in Figure 4-41.

**Figure 4-41** A circular path after bounding rectangle changed



In this example, the `GXSetShapeBounds` function actually alters the geometry of the original shape. If you call

```
GXGetShapeArea(aPathShape, 0, &theArea);
```

the area returned in the `theArea` parameter reflects the area of the new, smaller, geometry.

However, if you set the `gxMapTransformShape` shape attribute of the path shape before setting the shape bounds, QuickDraw GX moves and resizes the shape by changing the information in the shape's transform—not by changing the geometric points of the shape's geometry. In this case, calling the `GXGetShapeArea` function, which examines only a shape's geometry and ignore its transform mapping, results in the area of the original geometry. The result of declaring a new bounding rectangle and then calling

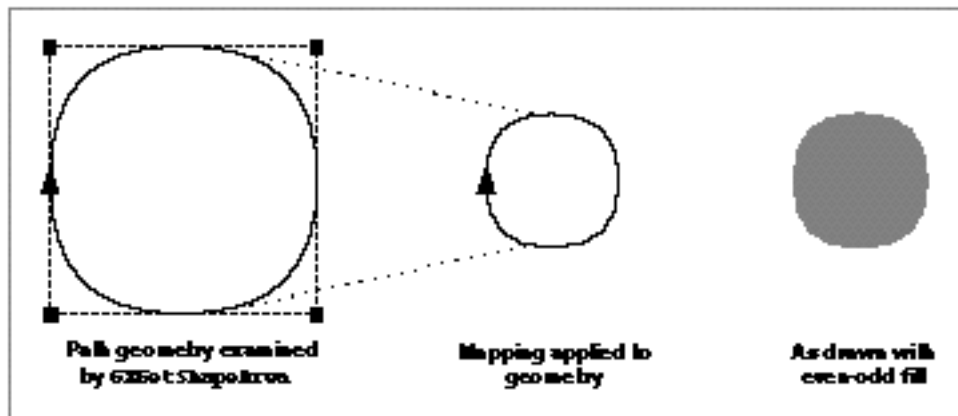
```
GXSetShapeAttributes(aPathShape,
                    GXGetShapeAttributes(aPathShape) |
                    gxMapTransformShape);
```

```
GXSetShapeBounds(aPathShape, &newBounds);
```

```
GXGetShapeArea(aPathShape, 0, &theArea);
```

is shown in Figure 4-42.

**Figure 4-42** A path shape with a transform mapping



For more information about the `GXSetShapeBounds` function, see page 4-92. For more information about the `GXGetShapeBounds` function, see page 4-90.

For more information about the `gxMapTransformShape` shape attribute, see the chapter “Shape Objects” and the chapter “Transform Objects” of *Inside Macintosh: QuickDraw GX Objects*.

## Insetting Shapes

---

Whereas the `GXSetShapeBounds` function, illustrated in the previous section, provides a way to scale a shape, the `GXInsetShape` function provides a way to resize a shape relative to its original contours.

The sample function in Listing 4-9 creates a curve shape to use as an example.

---

**Listing 4-9**      Creating a tight curve shape

```
void CreateATightCurve(void)
{
    gxShape curveShape;

    const gxCurve tightCurveGeometry = {ff(90), ff(200),
                                         ff(110), ff(0),
                                         ff(120), ff(200)};

    curveShape = GXNewCurve(&tightCurveGeometry);
    GXSetShapeFill(curveShape, gxOpenFrameFill);

    GXDrawShape(curveShape);
    GXDisposeShape(curveShape);
}
```

The result of this function is shown in Figure 4-43.

---

**Figure 4-43** A tight curve



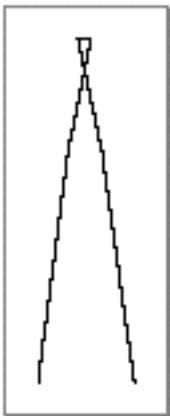
If you apply

```
GXInsetShape(curveShape, ff(10));
```

to this curve, the function insets the curve a distance of 10.0 points from the original geometry, as shown in Figure 4-44. The resulting shape is a path shape with 16 geometric points.

---

**Figure 4-44** An inset curve shape



You can use a shape's curve error to control the number of geometric points in the shape resulting from the inset operation. The result of the `GXInsetShape` function has no two consecutive points closer than the distance indicated by the shape's curve error.

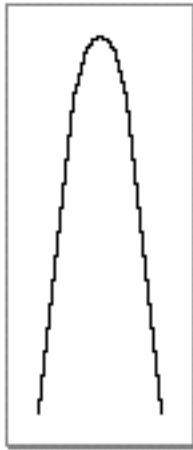
The `GXInsetShape` function considers the contour direction when calculating an inset contour. For example, if you use

```
GXReverseShape( curveShape, 1 );
```

to reverse the direction of the curve from Listing 4-9 before you inset the curve, the resulting path shape is actually outset from the original curve, as shown in Figure 4-45.

---

**Figure 4-45** An outset curve



The contours created by the `GXInsetShape` function lie to the right of the original contours if you specify a positive distance and to the left of the original contours if you specify a negative distance. You can alter this behavior by setting the auto-inset style attribute, as described in Chapter 3, “Geometric Styles,” in this book.

For more information about the `GXInsetShape` function, see page 4-94.

## Determining Whether Two Shapes Touch

---

QuickDraw GX provides three functions to help you determine whether the areas of two shapes touch—that is, whether they intersect, even at a single point.

- n The `GXTouchesRectanglePoint` function determines whether a point lies within the boundaries of a rectangle.
- n The `GXTouchesBoundsShape` function determines whether the area covered by a shape touches the area covered a rectangle.
- n The `GXTouchesShape` function determines whether one shape touches another shape

This section shows examples of using the `GXTouchesShape` function. The sample function in Listing 4-10 defines a rectangle and a circular path shape to use for these examples.

---

**Listing 4-10** Creating a rectangle and a circular path shape

```
void CreateBoxedCircle(void)
{
    gxShape aLargeCircle;

    static gxRectangle largeBounds = {ff(50), ff(50),
                                      ff(150), ff(150)};

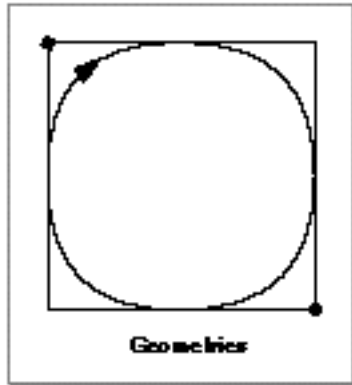
    static long largeCircleGeometry[] = {1,/* number of contours */
                                         4,/* number of points */
                                         0xF0000000, /* 1111 ... */
                                         ff(50), ff(50), /* off */
                                         ff(150), ff(50), /* off */
                                         ff(150), ff(150),/* off */
                                         ff(50), ff(150)};/* off */

    aLargeCircle = GXNewPaths((gxPaths *) largeCircleGeometry);
    GXSetShapeFill(aLargeCircle, gxClosedFrameFill);

    GXDrawRectangle(&largeBounds, gxClosedFrameFill);
    GXDrawShape(aLargeCircle);
    GXDisposeShape(aLargeCircle);
}
```

The result of this function is shown in Figure 4-46.

**Figure 4-46** A rectangle containing a circular path



You can call the `GXTouchesBoundsShape` function to test whether the rectangle and the path shape defined in Listing 4-10 touch:

```
GXTouchesBoundsShape(&largeBounds, aLargeCircle)
```

This function call returns `true`; the area of the rectangle does intersect the area of the path.



## Geometric Operations

When calculating whether a rectangle and a shape intersect, the `GXTouchesBoundsShape` function assumes that the rectangle has an even-odd shape fill. The following code defines another, smaller, path shape to test for intersection with the rectangle defined in Listing 4-10:

```
gxShape aSmallCircle;

static long smallCircleGeometry[] = {1, /* number of contours */
                                     4, /* number of points */
                                     0xF0000000,
                                     ff(65), ff(65), /* off */
                                     ff(135), ff(65), /* off */
                                     ff(135), ff(135), /* off */
                                     ff(65), ff(135)}; /* off */

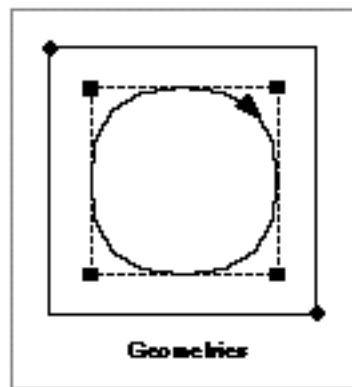
aSmallCircle = GXNewPaths((gxPaths *) smallCircleGeometry);
GXSetShapeFill(aSmallCircle, gxClosedFrameFill);
```

The call

```
GXTouchesBoundsShape(&largeBounds, aSmallCircle);
```

returns `true` because the smaller path shape touches the area contained by the rectangle, as shown in Figure 4-47.

**Figure 4-47** A rectangle that touches a circular path shape



## Geometric Operations

The `GXTouchesBoundsShape` function returns `true` even if the rectangle and the path shape share only an edge or even a single point. For example, if you move the small circle to the right by a distance of 85.0 points by calling

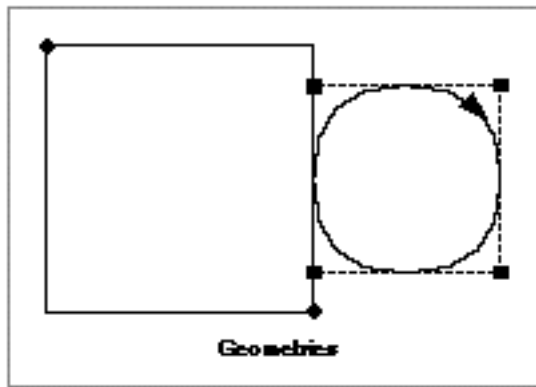
```
GXMoveShape(aSmallCircle, ff(85), 0);
```

as depicted in Figure 4-48, then the call

```
GXTouchesBoundsShape(&largeBounds, aSmallCircle)
```

still returns `true`.

**Figure 4-48** A rectangle and a circular path touching at a single point



The `GXTouchesShape` function works similarly to the `GXTouchesBoundsShape` function, but it determines whether any two shapes intersect.

As an example, if you give the path shapes defined earlier in this section the even-odd shape fill by calling

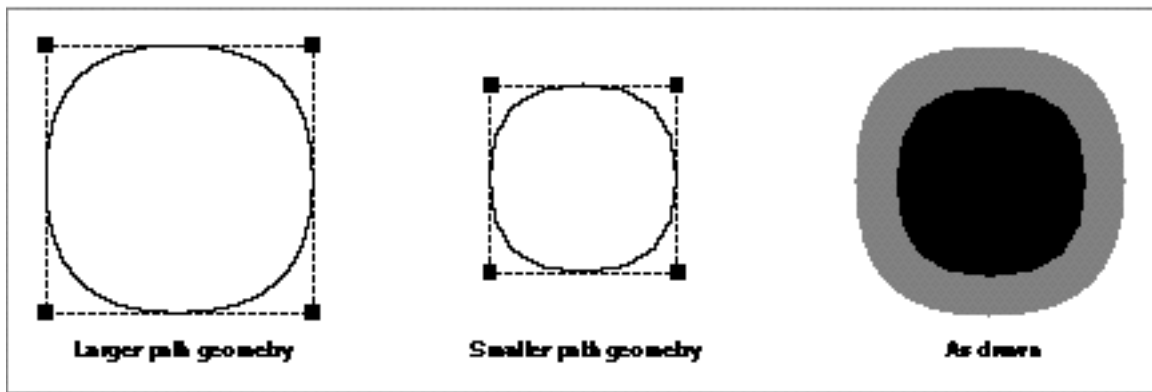
```
GXSetShapeFill(aLargeCircle, gxEvenOddFill);
GXSetShapeFill(aSmallCircle, gxEvenOddFill);
```

then the call

```
GXContainsShape(aLargeCircle, aSmallCircle)
```

returns `true`; the small path intersects the area contained in the large path, as shown in Figure 4-49.

**Figure 4-49** A large circular path shape touching a smaller circular path shape



For information about the `GXTouchesRectanglePoint` function, see page 4-96. For more information about the `GXTouchesBoundsShape` function and the `GXTouchesShape` function, see page 4-97 and page 4-98, respectively.

## Determining Whether One Shape Contains Another

---

QuickDraw GX also provides three functions to help you determine whether one area contains another:

- n The `GXContainsRectangle` function determines whether one rectangle contains another.
- n The `GXContainsBoundsShape` function determines whether the area covered by a rectangle contains the area covered by a shape.
- n The `GXContainsShape` function determines whether the area covered by one shape contains the area covered by another shape.

The sample function in Listing 4-11 creates a small circular path and a larger, donut-shaped path to test for containment.

---

**Listing 4-11** Creating a path shape with two contours and a smaller concentric rectangle shape

```
void CreateMultiplePaths(void)
{
    gxShape twoCircleShape, smallSquareShape;

    static rectangle smallSquareGeometry[] = {ff(90), ff(90),
                                              ff(110), ff(110)};

    static long twoCircleGeometry[] = {2, /* # of contours */
                                       4, /* # of points */
                                       0xF0000000, /* 1111 ... */
                                       ff(50), ff(50),
                                       ff(150), ff(50),
                                       ff(150), ff(150),
                                       ff(50), ff(150),
                                       4, /* # of points */
                                       0xF0000000, /* 1111 ... */
                                       ff(65), ff(65),
                                       ff(65), ff(135),
                                       ff(135), ff(135),
                                       ff(135), ff(65)};

    twoCircleShape = GXNewPaths((gxPaths *) twoCircleGeometry);
    GXSetShapeFill(twoCircleShape, gxEvenOddFill);
}
```

## Geometric Operations

```

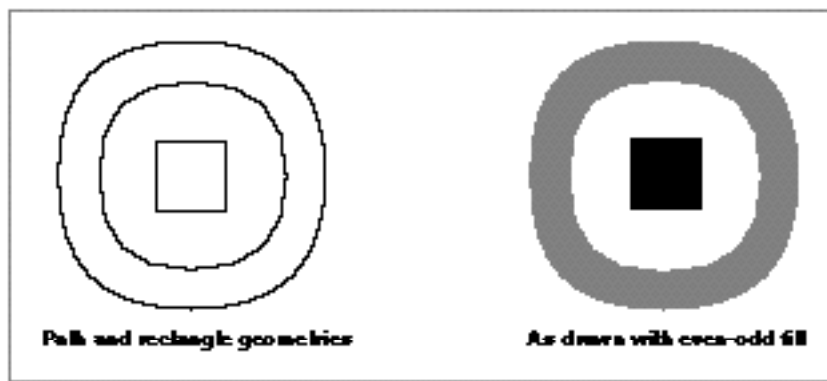
    smallSquareShape = GXNewRectangle(&smallSquareGeometry);
    GXSetShapeFill(smallSquareShape, gxEvenOddFill);

    GXDrawShape(twoCircleShape);
    GXDisposeShape(twoCircleShape);
    GXDrawShape(smallSquareShape);
    GXDisposeShape(smallSquareShape);
}

```

The results of this sample function are shown in Figure 4-50.

**Figure 4-50** A path shape with two contours and a smaller concentric rectangle shape



Since the `GXContainsShape` function considers shape fill when calculating whether one shape contains another, the following function call:

```
GXContainsShape(twoCircleShape, smallerCircleShape);
```

returns `false`; the area covered by the larger path does not contain the area covered by the smaller path.

For more information about the `GXContainsShape` function, see page 4-103.

The functions `GXContainsRectangle` and `GXContainsBoundsShape` work similarly to the `GXContainsShape` function, except the input parameters to these functions are rectangle geometries, rather than shapes. The `GXContainsRectangle` function compares two rectangle geometries and the `GXContainsBoundsShape` compares a rectangle geometry to a shape.

For more information about the `GXContainsRectangle` function and the `GXContainsBoundsShape` function, see page 4-100 and page 4-101, respectively.

## Performing Geometric Arithmetic With Shapes

---

QuickDraw GX provides six arithmetic operations you can apply to geometric shapes: union, intersection, difference, reverse difference, exclusion, and inversion.

To illustrate these operations, the sample function in Listing 4-12 creates two shapes: a diamond-shaped polygon and a circular path.

---

**Listing 4-12** Creating a diamond-shaped polygon and a circular path that intersect

```
void CreateDiamondAndCircle(void)
{
    gxShape  diamondShape, circleShape;

    static long circleGeometry[] = {1, /* number of contours */
                                    4, /* number of points */
                                    0xF0000000, /* 1111 ... */
                                    ff(100), ff(100), /* off */
                                    ff(200), ff(100), /* off */
                                    ff(200), ff(200), /* off */
                                    ff(100), ff(200)}; /* off */

    static long diamondGeometry[] = {1, /* number of contours */
                                     4, /* number of points */
                                     ff(50), ff(150),
                                     ff(100), ff(100),
                                     ff(150), ff(150),
                                     ff(100), ff(200)};

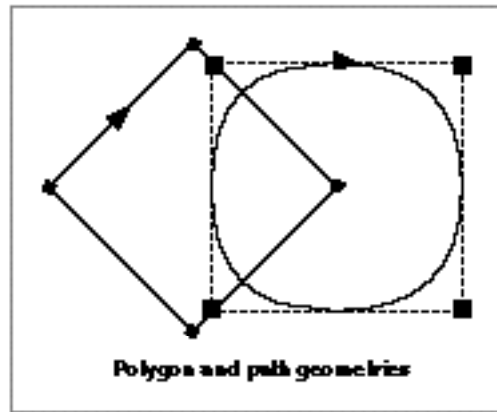
    diamondShape = GXNewPolygons((gxPolygons *) diamondGeometry);

    circleShape = GXNewPaths((gxPaths *) circleGeometry);

    GXDrawShape(diamondShape);
    GXDisposeShape(diamondShape);
    GXDrawShape(circleShape);
    GXDisposeShape(circleShape);
}
```

The resulting shapes are shown in Figure 4-51.

**Figure 4-51** A diamond-shaped polygon geometry and a circular path geometry



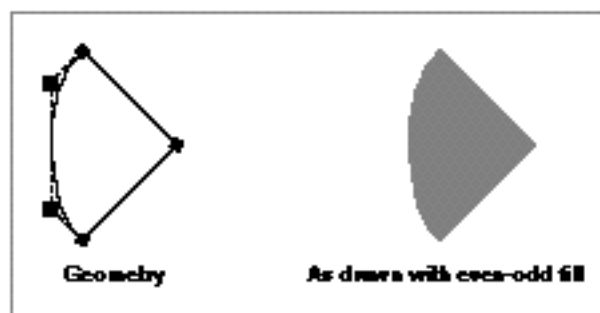
The `GXIntersectShape` function finds the area common to two shapes. This function takes two parameters—the shapes to intersect—and stores the result in the first parameter.

If you apply the `GXIntersectShape` function to the diamond-shaped polygon and the circular path from Listing 4-12 by calling

```
GXIntersectShape(diamondShape, circleShape);
GXDrawShape(diamondShape);
```

you get the resulting shape shown in Figure 4-52.

**Figure 4-52** The intersection of a diamond-shaped polygon and a circular path



**Implementation Note**

Due to an implementation limit with QuickDraw GX version 1.0, you can find the intersection of two framed shapes only if the shapes are points, lines, or curves. You can, however, find the intersection of a framed shape and a filled shape; the intersection is the part of the framed shape contained in the filled shape. In this case, the target shape must be the framed shape and the operand shape must be the filled shape.  $\cup$

You can find the intersection of two rectangle geometries—without having to encapsulate those geometries into shapes—using the `GXIntersectRectangle` function, which is described on page 4-105. Similarly, you can find the union of two rectangle geometries (which is considered to be the smallest rectangle that contains them both) using the `GXUnionRectangle` function, which is described on page 4-106.

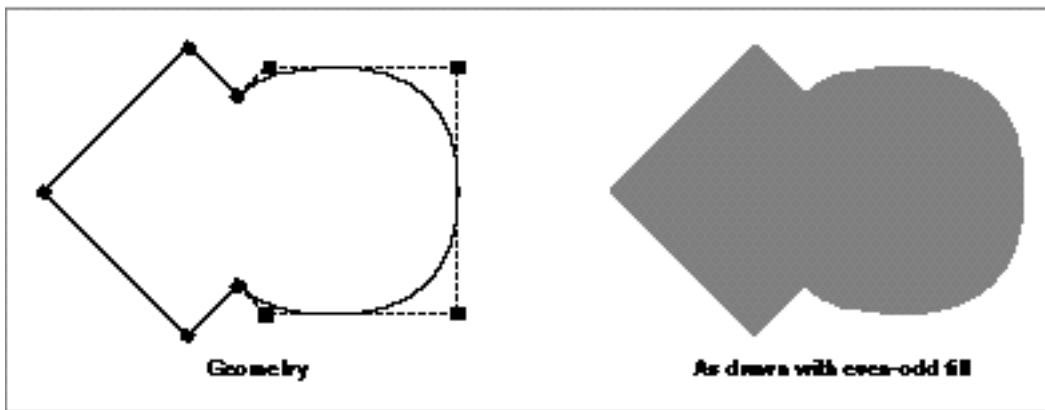
The `GXUnionShape` function combines the areas covered by two shapes. This function also takes two parameters—the shapes to combine—and stores the result in the first parameter.

If you apply the `GXUnionShape` function to the diamond-shaped polygon and the circular path from Listing 4-12 by calling

```
GXUnionShape(diamondShape, circleShape);
GXDrawShape(diamondShape);
```

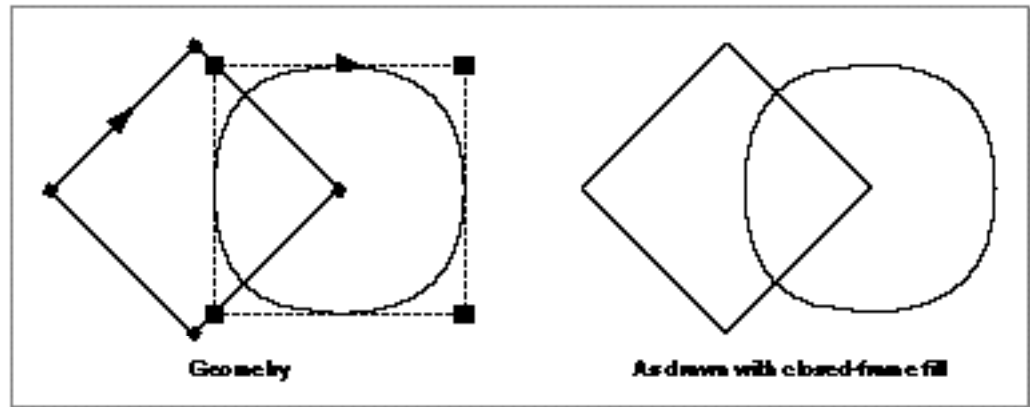
you get the resulting shape shown in Figure 4-53.

**Figure 4-53** The union of a diamond-shaped polygon and a circular path



Although you cannot find the union of a framed shape and a solid shape, you can find the union of two framed shapes. If the diamond-shaped polygon and the circular path from Listing 4-12 had closed-frame fills, the resulting union would appear as shown in Figure 4-54.



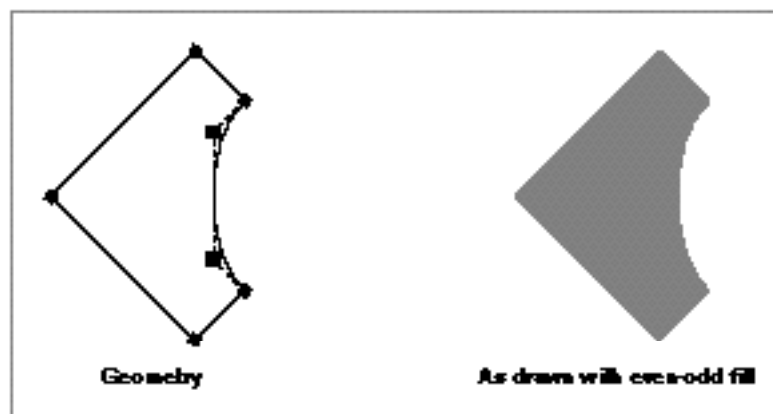
**Figure 4-54** The union of a framed diamond-shaped polygon and a circular path

The `GXDifferenceShape` function subtracts the area of one shape from the area of another. This function takes two parameters—the shape to subtract from and the shape to subtract—and stores the result in the first parameter.

If you apply the `GXDifferenceShape` function to the diamond-shaped polygon and the circular path from Listing 4-12 by calling

```
GXDifferenceShape(diamondShape, circleShape);
GXDrawShape(diamondShape);
```

you get the resulting shape shown in Figure 4-55.

**Figure 4-55** The result of subtracting a circular path from a diamond-shaped polygon

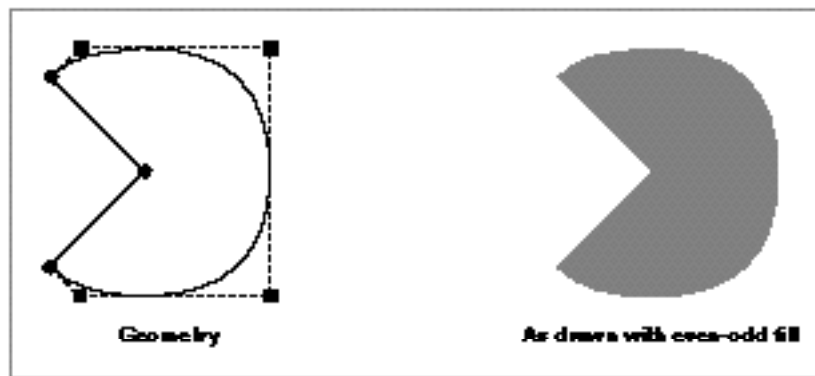
The `GXReverseDifferenceShape` function is similar to the `GXDifferenceShape` function, except the `GXReverseDifferenceShape` function subtracts the first parameter from the second parameter. Like `GXDifferenceShape`, it also stores the result in the first parameter.

If you apply the `GXReverseDifferenceShape` function to the diamond-shaped polygon and the circular path from Listing 4-12 by calling

```
GXReverseDifferenceShape(diamondShape, circleShape);
GXDrawShape(diamondShape);
```

you get the resulting shape shown in Figure 4-56.

**Figure 4-56** The result of subtracting a diamond-shaped polygon from a circular path



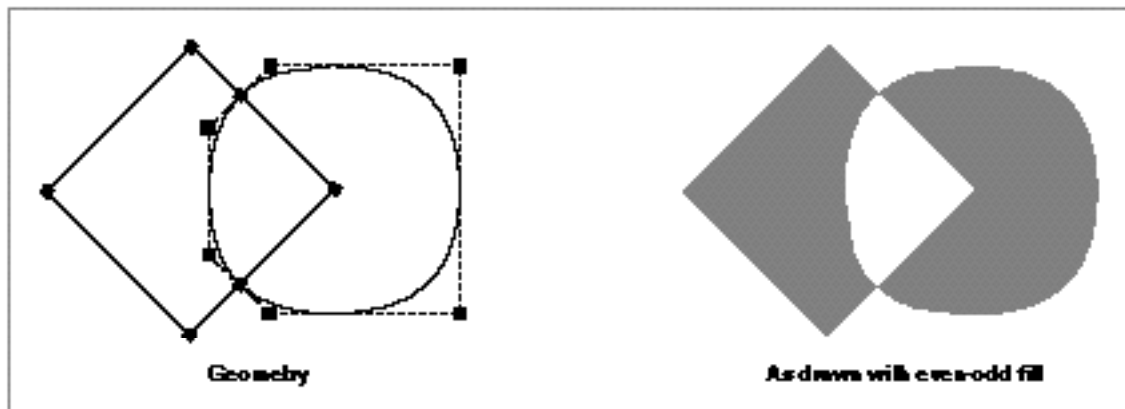
The `GXExcludeShape` function performs the exclusive-OR operation on the areas of two shapes—that is, it finds the area that is covered by one shape or the other, but not by both shapes.

If you apply the `GXExcludeShape` function to the diamond-shaped polygon and the circular path from Listing 4-12 by calling

```
GXExcludeShape(diamondShape, circleShape);
GXDrawShape(diamondShape);
```

you get the resulting shape shown in Figure 4-57.

**Figure 4-57** The result of the exclusive-OR operation on a polygon and a path



Finally, QuickDraw GX provides the `GXInvertShape` function which inverts the area covered by a shape—that is, the resulting shape covers all of the area not covered by the original shape. This function takes one parameter—the shape to invert—and stores the result in this parameter.

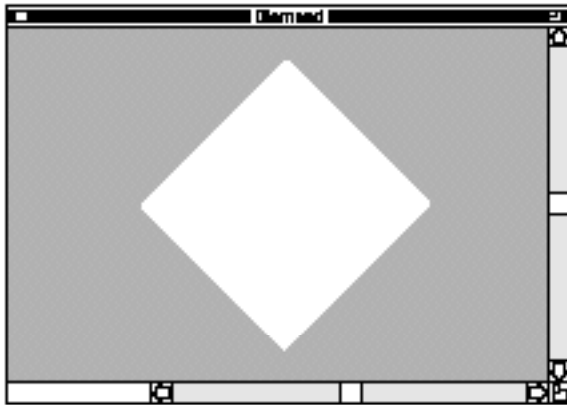
## Geometric Operations

If you apply the `GXInvertShape` function to the diamond-shaped polygon from Listing 4-12 by calling

```
GXInvertShape(diamondShape);
```

you get the resulting shape shown in Figure 4-58. Notice that this shape extends to the full extent of its view port.

**Figure 4-58** An inverted diamond



For more information about these arithmetic operations, see the function descriptions in “Performing Geometric Arithmetic With Shapes” beginning on page 4-104.

## Geometric Operations Reference

---

QuickDraw GX allows you to determine geometric information about and perform geometric manipulations on shapes. This section describes the data types and functions that are related to these operations.

### Constants and Data Types

---

This section describes the enumeration QuickDraw GX uses to specify information about contour direction.

#### Contour Directions

---

QuickDraw GX assigns a contour direction to every contour in a shape: contours are either clockwise or counterclockwise.

Contour directions are specified by the `gxContourDirections` enumeration, which is defined as follows:

```
enum gxContourDirections {
    gxCounterclockwiseDirection,
    gxClockwiseDirection
};

typedef long gxContourDirection;
```

**Constant descriptions**

`gxCounterclockwiseDirection`

A counterclockwise contour direction.

`gxClockwiseDirection`

A clockwise contour direction.

For more information about the contours of the various geometric shapes, see Chapter 2, “Geometric Shapes,” in this book.

For more information about contour direction, see “Contours and Contour Direction” beginning on page 4-4.

For more information about how QuickDraw GX uses contour direction, see the description of the `GXReverseShape` function on page 4-70.

## Functions

---

QuickDraw GX provides functions you can call to perform geometric operations on geometric shapes and their geometries. This section includes descriptions of the functions that allow you to

- n determine and alter the contour direction of a shape's contours
- n reduce and simplify a shape's geometry
- n incorporate style information into a shape's geometry
- n obtain geometric information about a shape's geometry
- n determine and alter the bounding rectangle of a shape
- n inset a shape's geometry
- n perform geometric arithmetic on shapes

## Determining and Reversing Contour Direction

---

The contours of geometric shapes have a contour direction: clockwise or counterclockwise. The following factors determine the contour direction of a contour:

- n the order and position of the geometric points that make up the contour
- n the contour's relative position to other contours in the shape if the shape has multiple contours

For a discussion of geometric points, see Chapter 2, "Geometric Shapes," in this book.

The `GXGetShapeDirection` function allows you to determine the contour direction of a specified contour of a shape.

The `GXReverseShape` function allows you to reverse the order of the geometric points that define a contour and therefore reverse the contour's direction.

## GXGetShapeDirection

---

You can use the `GXGetShapeDirection` function to determine whether a contour has a clockwise or counterclockwise direction.

```
gxContourDirection GXGetShapeDirection(gxShape source,
                                         long contour);
```

`source`        A reference to the shape containing the contour.

`contour`       The contour index of the contour whose direction you want to determine.

## Geometric Operations

*function result* The direction of the contour, either `gxClockwiseDirection` or `gxCounterclockwiseDirection`.

## DESCRIPTION

The `GXGetShapeDirection` function indicates whether QuickDraw GX considers the contour indicated by the `contour` parameter of the shape indicated by the `source` parameter to be clockwise or counterclockwise. You can use this information to determine how QuickDraw GX will draw a shape that has the `gxInsideFrameStyle` or `gxOutsideFrameStyle` style attribute set, or how the `GXInsetShape` function affects a shape.

For empty and full shapes, this function posts the error `graphic_type_does_not_have_multiple_contours`.

For point shapes and line shapes, this function always returns `gxClockwiseDirection`. Although the order of the geometric points in a line shape's geometry does not affect the result of this function, it may affect how QuickDraw GX draws the line. For example, if the line is dashed, the order of the geometric points determines the end of the line at which dashing begins. Also, if the line has a pen width and the `gxInsideFrameStyle` or `gxOutsideFrameStyle` style attribute is set, the order of the geometric points determines the side of the geometry on which QuickDraw GX draws the line.

For rectangle shapes, the result of this function is determined by examining which corners are specified by the geometric points. For a rectangle whose geometry includes the upper-left point and the lower-right point, this function returns `gxClockwiseDirection`, regardless of the order of the two points in the geometry. For a rectangle whose geometry includes the upper-right point and the lower-left point, this function returns `gxCounterclockwiseDirection`, again regardless of the order of the two points.

For curve shapes, polygon shapes, and path shapes, reversing the order of any contour's geometric points reverses the result of this function.

If you provide a source shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts the error <code>graphic_type_does_not_have_multiple_contours</code>
picture	Posts the error <code>graphic_type_does_not_have_multiple_contours</code>
text	Posts the error <code>illegal_type_for_shape</code>
glyph	Posts the error <code>illegal_type_for_shape</code>
layout	Posts the error <code>illegal_type_for_shape</code>

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>contour_is_less_than_zero</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>graphic_type_does_not_have_multiple_contours</code>	(debugging version)

**Warnings**

<code>contour_out_of_range</code>
-----------------------------------

## SEE ALSO

For examples using this function, see “Determining and Reversing Contour Direction” beginning on page 4-23.

To reverse the order of geometric points in a shape’s geometry, use the `GXReverseShape` function, described in the next section.

For a description of the `GXInsetShape` function, see page 4-94. For a description of the `gxInsideFrameStyle` and `gxOutsideFrameStyle` style attributes, see Chapter 3, “Geometric Styles,” in this book.

**GXReverseShape**

---

You can use the `GXReverseShape` function to reverse the order of the geometric points in a shape’s contour.

```
void GXReverseShape(gxShape target, long contour);
```

<code>target</code>	A reference to the shape containing the contour.
<code>contour</code>	The number of the contour you want to reverse. You may specify a value of 0 for this parameter to indicate all contours.

## DESCRIPTION

The `GXReverseShape` function reverses the order of the geometric points of the contour specified by the `contour` parameter in the shape specified by the `target` parameter.

If you specify a value of 0 for the `contour` parameter, this function reverses the order of the geometric points in each contour of the target shape, but does not affect the order of the contours themselves. If the target shape is a rectangle shape, this function converts it to a polygon shape before reversing the direction.



You can use this function to control how QuickDraw GX

- n draws shapes with a winding shape fill
- n draws shapes with the `gxInsideFrameStyle` or `gxOutsideFrameStyle` style attribute set
- n places dashes on a dashed contour
- n insets shape geometries

If you provide a target shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts the error <code>illegal_type_for_shape</code>
picture	Posts the error <code>illegal_type_for_shape</code>
text	Posts the error <code>illegal_type_for_shape</code>
glyph	Posts the error <code>illegal_type_for_shape</code>
layout	Posts the error <code>illegal_type_for_shape</code>

#### ERRORS, WARNINGS, AND NOTICES

##### Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>contour_is_less_than_zero</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)

##### Warnings

<code>contour_out_of_range</code>
<code>shape_access_not_allowed</code>

#### SEE ALSO

For examples using this function, see “Determining and Reversing Contour Direction” beginning on page 4-23.

To determine the direction of a contour, use the `GXGetShapeDirection` function, described on page 4-68.

For a discussion of geometric points, see the Chapter 2, “Geometric Shapes.”

For a discussion of contour direction, see “Contours and Contour Direction” beginning on page 4-4.

For a discussion of dashes and the `gxInsideFrameStyle` or `gxOutsideFrameStyle` style attributes, see the Chapter 3, “Geometric Styles,” in this book.

For a description of the `GXInsetShape` function, see page 4-94.

## Breaking Shape Contours

---

Each contour of a polygon or path shape can be made up of many parts: each polygon contour can contain many lines and each path contour can contain many lines and curves.

The `GXBreakShape` function allows you to specify a geometric point at which to break a single contour into two contours.

## GXBreakShape

---

You can use the `GXBreakShape` function to break a single contour into two contours.

```
void GXBreakShape(gxShape target, long index);
```

**target**            A reference to the shape containing the contour to break.

**index**            The geometry index of the point at which to break the contour.

### DESCRIPTION

The `GXBreakShape` function breaks an existing contour into two contours at a specified geometric point.

This function can convert the shape type of the target shape. For example, you can break line shapes at their first point by specifying a geometry index of 1 for the `index` parameter. The result is a polygon shape with two contours: the first contour is empty and the second contour is the original line. If you specify a value of 2 for the `index` parameter, this function posts the notice `shape_already_broken`, and the original line is unaffected.

Similarly, you can break curve shapes at their first point; the result is a path shape with two contours. You can also break curve shapes at the off-curve control point by specifying a value of 2 for the `index` parameter. The resulting path shape has two contours: the first contour ends at the off-curve control point, and the second contour begins at the off-curve control point. You must add on-curve geometric points at the end of the first contour and the beginning of the second contour before drawing this path shape.

The function affects polygon and path shapes in the following ways:

- n If the geometry index you specify corresponds to the first point of a contour, this function inserts an empty contour into the shape before the specified point.
- n If the geometry index you specify corresponds to the last point of a contour, this function posts a `shape_already_broken` notice, and the original shape is unaffected.
- n If the geometric index you specify corresponds to a point between the first and last points of a contour, this function breaks the existing contour into two contours. The specified point becomes the first point of the new second contour.

## Geometric Operations

For empty, full, and point shapes, this function posts the error `graphic_type_does_not_contain_points`. For rectangle shapes, this function posts a `rectangles_cannot_be_inserted_into` notice.

If you provide a target shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts the error <code>graphic_type_does_not_contain_points</code>
picture	Posts the error <code>graphic_type_does_not_contain_points</code>
text	Posts the error <code>illegal_type_for_shape</code>
glyph	Posts the error <code>illegal_type_for_shape</code>
layout	Posts the error <code>illegal_type_for_shape</code>

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>index_is_less_than_one</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)
<code>rectangles_cannot_be_inserted_into</code>	(debugging version)
<code>graphic_type_does_not_contain_points</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)

**Warnings**

<code>shape_access_not_allowed</code>
<code>index_out_of_range</code>
<code>contour_out_of_range</code>

**Notices (debugging version)**

<code>shape_already_broken</code>
-----------------------------------

## SEE ALSO

For an example using this function, see “Breaking Shape Contours” beginning on page 4-28.

For a discussion of geometric points, see Chapter 2, “Geometric Shapes,” in this book. For other methods of breaking contours, see the shape-editing functions also described in that chapter.

To learn how this function works for the typographic shapes, see *Inside Macintosh: QuickDraw GX Typography*.

## Reducing and Simplifying Shapes

---

The geometries of QuickDraw GX shapes can contain many unnecessary or complicating elements:

- n duplicate geometric points
- n unnecessary colinear geometric points
- n crossed contours
- n overlapping contours
- n unnecessary contour breaks
- n a more complex shape type than necessary

The `GXReduceShape` function eliminates unnecessary geometric points.

The `GXSimplifyShape` function eliminates crossed contours, overlapping contours, unnecessary contour breaks, and also sets a shape's shape type to the simplest type necessary to describe the shape's geometry.

## GXReduceShape

---

You can use the `GXReduceShape` function to remove unnecessary geometric points from a polygon or path contour.

```
void GXReduceShape(gxShape target, long contour);
```

target	A reference to the polygon or path shape containing the contour whose unnecessary geometric points you want to eliminate.
contour	The index of the contour you want to reduce. You may specify a value of 0 for this parameter to indicate all contours.

### DESCRIPTION

The `GXReduceShape` function removes unnecessary geometric points from the contour indicated by the `contour` parameter of the shape indicated by the `target` parameter. The geometric points removed by this function include both duplicate and colinear geometric points. Duplicate geometric points are sequential geometric points in the same contour with the same (x, y) coordinate pair. Colinear geometric points are sequential geometric points that fall on the same line as the preceding and the subsequent geometric point. Although this function may affect the geometry of a shape, the resulting shape appears the same as the original shape when drawn, unless the curve error of the target shape is nonzero.

**Note**

Under certain circumstances, the `GXReduceShape` function actually increases the number of geometric points used to define a shape. For path shapes, the number of geometric points in the resulting shape can be up to one third more than the number of points in the original shape. Even in this case, the resulting shape appears the same as the original shape when drawn. <sup>u</sup>

The `GXReduceShape` function does consider the curve error of the target shape when selecting which geometric points to remove. If the distance between a point and a neighboring point is less than that indicated by the curve error, the `GXReduceShape` function considers them to be duplicate points. If you specify a target shape with a nonzero curve error, the resulting shape may draw differently than the original shape—the greater the curve error, the more drastic the difference may be. For shapes with many points within a distance of less than that indicated by the curve error, the resulting shape can sometimes degenerate to a surprising result.

The shape fill of the target shape can also affect the results of this function. For example, if the first point and the last point of a contour are the same geometric point, this function removes the last point if the target shape has a closed-frame fill or any of the solid fills. However, if the target shape has an open-frame fill, this function does not remove the last point.

Similarly, if one or more of the points at the end of a contour is colinear with one or more points at the beginning of that contour, this function considers them all to lie on the same line if the target shape has a closed-frame fill or any of the solid fills and the unnecessary points are removed—even the first and last point of the original contour can be removed. However, if the target shape has an open-frame fill, the first and the last points of a contour are never removed.

This function operates only within individual contours; it never combines contours or compares points from different contours. Also, this function does not convert between shape types. The resulting shape always has the same shape type as the original shape.

If you specify a source shape that is not a polygon or a path shape, this function posts the error `graphic_type_cannot_be_reduced`.

If you provide a target shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts the error <code>graphic_type_cannot_be_reduced</code>
picture	Posts the error <code>graphic_type_cannot_be_reduced</code>
text	Posts the error <code>graphic_type_cannot_be_reduced</code>
glyph	Posts the error <code>graphic_type_cannot_be_reduced</code>
layout	Posts the error <code>graphic_type_cannot_be_reduced</code>

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`  
`shape_is_nil`  
`size_of_path_exceeds_implementation_limit`  
`number_of_points_exceeds_implementation_limit`  
`graphic_type_cannot_be_reduced` (debugging version)

**Warnings**

`unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve` (debugging version)  
`contour_out_of_range`  
`shape_access_not_allowed`

**SEE ALSO**

For examples using this function, see “Eliminating Unnecessary Geometric Points” beginning on page 4-30.

For more information about reduced and simplified shapes, see “Reducing and Simplifying Shape Geometries” beginning on page 4-9.

For a discussion of geometric points and contours, see Chapter 2, “Geometric Shapes,” in this book.

**GXSimplifyShape**

---

You can use the `GXSimplifyShape` function to eliminate from a shape any unnecessary contour breaks, contour crossings, and internal contour loops.

```
void GXSimplifyShape(gxShape target);
```

`target`      A reference to the shape you want to simplify.

**DESCRIPTION**

The `GXSimplifyShape` function performs operations on the geometry of the shape specified by the `target` parameter and simplifies the description of the shape, sometimes changing the shape type, without affecting how the shape is drawn.

Most importantly, the resulting shape has no crossed contours. The `GXSimplifyShape` function adds geometric points and changes contour directions to redefine the shape's geometry so that no contour crosses over itself or any other contour.

This function also removes unnecessary contour breaks. If the last point of one contour is identical to the first point of the next contour, this function combines the two contours into a single contour.

**Note**

Under certain circumstances, the `GXSimplifyShape` function actually increases the number of geometric points and the number of contours used to define a shape. However, the simplified shape still appears the same as the original shape when drawn. <sup>u</sup>

If the geometry of the original shape can be expressed as a geometry of a simpler shape type, this function converts the shape to the simpler type. For example, if the shape referenced by the `target` parameter is a polygon, but the geometry of that polygon defines a simple square, the `GXSimplifyShape` function converts the shape to a rectangle type and redefines the geometry as appropriate. As another example, a path shape with no curved contours is converted to a polygon shape type.

The shape fill of the target shape also affects the simplifications. For example, if the target shape has two circular, concentric contours (an inner contour and an outer contour) and both contours have the same contour direction, the following occurs:

- n If the shape has a winding shape fill, the inner contour does not affect how the shape is drawn. In this case, the `GXSimplifyShape` function removes the inner contour.
- n If the shape has an even-odd shape fill, the inner contour does affect how the shape is drawn. In this case, the `GXSimplifyShape` function maintains the inner contour, but it reverses the direction of that contour.

As a result of these simplifications, changing the shape fill of a simplified shape from winding fill to even-odd fill or from even-odd fill to winding fill does not affect the appearance of the shape when drawn.

## Geometric Operations

If you provide a target shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Chooses smaller color set if possible, if the pixel size is 1, 2, 4, or 8; posts the notice <code>shape_already_in_simple_form</code> if the pixel size is 16 or 32; converts to a rectangle shape if every pixel in the bitmap has the same color
picture	Posts the notice <code>shape_already_in_simple_form</code>
text	Simplifies to the empty shape if appropriate; posts the notice <code>shape_already_in_simple_form</code> otherwise
glyph	Simplifies to the empty shape, a text shape, or a simpler glyph shape as appropriate; posts the notice <code>shape_already_in_simple_form</code> if no simplification possible
layout	Simplifies to the empty shape if appropriate; posts the notice <code>shape_already_in_simple_form</code> otherwise

## ERRORS, WARNINGS, AND NOTICES

**Errors**

```

out_of_memory
shape_is_nil
number_of_contours_exceeds_implementation_limit
number_of_points_exceeds_implementation_limit
size_of_path_exceeds_implementation_limit
size_of_polygon_exceeds_implementation_limit
shape_access_not_allowed
functionality_unimplemented                                     (debugging version)

```

**Warnings**

```

unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve
                                                                    (debugging version)

```

**Notices (debugging version)**

```

shape_already_in_simple_form

```

## SEE ALSO

For examples using this function, see “Simplifying Shapes” beginning on page 4-33.

For more information about simplified shapes, see “Reducing and Simplifying Shape Geometries” beginning on page 4-9.

For a discussion of geometric points and contours, see Chapter 2, “Geometric Shapes,” in this book.

To remove unnecessary geometric points but not perform other simplifications, use the `GXReduceShape` function, described on page 4-74.



## Incorporating Style Information Into Shape Geometries

---

QuickDraw GX requires that shapes used for certain purposes (caps, joins, dashes, patterns, and clips) be in primitive form—that is, their style modifications must be incorporated into their geometries. For example, the `GXSetShapeDash` function requires that the shape used for dashing be in primitive form; the `GXSetShapeCap`, `GXSetShapeJoin`, and `GXSetShapePattern` functions are similar.

For more information about the primitive form of shapes and for examples of functions that use shapes in their primitive form, see Chapter 3, “Geometric Styles,” in this book.

The `GXPrimitiveShape` function converts a shape to its primitive form, incorporating the modifications made by the shape’s style into the shape’s geometry.

## GXPrimitiveShape

---

You can use the `GXPrimitiveShape` function to convert a shape to its primitive form.

```
void GXPrimitiveShape(gxShape target);
```

`target`      A reference to the shape to convert to primitive form.

### DESCRIPTION

The `GXPrimitiveShape` function converts the shape referenced by the `target` parameter to its primitive form—that is, it changes the geometry, shape fill, and shape type of the target shape to incorporate the information from the original shape’s style (including pen width, dashes, joins, and so on).

For example, a horizontal line shape with a greater-than-zero pen width becomes a filled rectangle shape. A diagonal line shape with a greater-than-zero pen width becomes a filled polygon shape. A curve shape with a greater-than-zero pen width becomes a filled path shape. A framed shape dashed with rectangles becomes a polygon shape with multiple contours—each contour representing one of the original dashes.

For the geometric shapes, the shape resulting from this function can be a hairline shape or a solid-filled shape. In either case, the information from the style object is no longer necessary because it has been incorporated into the shape object itself.

### Implementation Note

In version 1.0 of QuickDraw GX, this function posts an error of `functionality_unimplemented` for picture shapes. <sup>u</sup>

The result of the `GXPrimitiveShape` function is not simplified, nor are its unnecessary geometric points removed. You may want to simplify or reduce the resulting shape by calling the `GXSimplifyShape` function or the `GXReduceShape` function.

The following table gives information about this function for each type of geometric shape:

Shape type	Action taken
empty	If the shape fill is the <code>noFill</code> shape fill, this function posts the notice <code>shape_already_in_primitive_form</code> . If the shape fill is either of the inverse shape fills, this function returns a full shape (unless the shape has a pattern, in which case the function returns the shape described by the pattern).
full	If the shape fill is the <code>noFill</code> shape fill or any of the inverse fills, this function returns an empty shape. Otherwise, the function posts the notice <code>shape_already_in_primitive_form</code> (unless the shape has a pattern, in which case the function returns the shape described by the pattern).
point	<p>If the shape fill is the <code>noFill</code> shape fill, this function returns an empty shape. If the pen width is greater than zero and the shape has a start cap, the function returns the start cap. If the pen width is zero or the shape has no start cap, the function returns an empty shape.</p> <p>If the shape has a pen width of zero, no start cap, and a solid pattern, the function returns a point as indicated by the pattern. If the shape has a framed pattern, the function posts the <code>clip_to_frame_shape_unimplemented</code> error. Bitmap patterns are ignored.</p> <p>If one of the grid-constraining attributes is set, this function constrains the point geometry to the grid.</p>
line	<p>If the shape fill is the <code>noFill</code> shape fill, this function returns an empty shape. If the pen width is greater than zero, the function returns a polygon shape (or a path shape depending on the start and end caps).</p> <p>If the pen width is zero and the shape has a solid pattern, the function returns a point or a line as indicated by the pattern. If the shape has a framed pattern, the function posts the <code>clip_to_frame_shape_unimplemented</code> error. Bitmap patterns are ignored.</p> <p>If one of the grid-constraining attributes is set, this function constrains the geometry to the grid.</p>

Shape type	Action taken
curve	<p>If the shape fill is the <code>noFill</code> shape fill, this function returns an empty shape. If the pen width is greater than zero, the function returns a path shape.</p> <p>If the pen width is zero and the shape has a solid pattern, the function returns a point, line, curve, or path line as indicated by the pattern. If the shape has a framed pattern, the function posts the <code>clip_to_frame_shape_unimplemented</code> error. Bitmap patterns are ignored.</p> <p>If one of the grid-constraining attributes is set, this function constrains the geometry to the grid.</p>
rectangle	<p>If the shape fill is the <code>noFill</code> shape fill, this function returns an empty shape. If the shape fill is one of the framed fills and the pen width is greater than zero, the function returns a polygon shape (or a path shape depending on the join shape).</p> <p>If the rectangle has a solid pattern, the function returns a point, line, or polygon as indicated by the pattern. If the rectangle is framed, has a pen width of zero, and has a framed pattern, the function posts the <code>clip_to_frame_shape_unimplemented</code> error. Bitmap patterns are ignored.</p> <p>If one of the grid-constraining attributes is set, this function constrains the geometry to the grid.</p>
polygon	<p>If the shape fill is the <code>noFill</code> shape fill or the shape has no contours, this function returns an empty shape. If the shape fill is one of the framed fills and the pen width is greater than zero, the function returns a polygon shape (or a path shape depending on the caps and join).</p> <p>If the shape has a solid pattern, the function returns a point, line, or polygon as indicated by the pattern. If the polygon is framed, has a pen width of zero, and has a framed pattern, the function posts the <code>clip_to_frame_shape_unimplemented</code> error. Bitmap patterns are ignored.</p> <p>If one of the grid-constraining attributes is set, this function constrains the geometry to the grid.</p>
path	<p>If the shape fill is the <code>noFill</code> shape fill or the shape has no contours, this function returns an empty shape. If the shape fill is one of the framed fills and the pen width is greater than zero, the function returns a path shape.</p> <p>If the shape has a solid pattern, the function returns a point, line, or polygon as indicated by the pattern. If the path is framed, has a pen width of zero, and shape has a framed pattern, the function posts the <code>clip_to_frame_shape_unimplemented</code> error. Bitmap patterns are ignored.</p> <p>If one of the grid-constraining attributes is set, this function constrains the geometry to the grid.</p>

## Geometric Operations

If you provide a target shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Constrains the bitmap position to an integer grid position if one of the grid-constraining attributes is set
picture	Posts the notice <code>shape_already_in_primitive_form</code>
text	Converts to glyph shape; constrains the text position to an integer grid position if one of the grid-constraining attributes is set; converts to an empty shape if appropriate
glyph	Constrains the glyph positions to integer grid positions if one of the grid-constraining attributes is set; eliminates any nil styles and complex styles; converts to an empty shape if appropriate
layout	Converts to a glyph shape; converts to an empty shape if appropriate

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>shape_access_not_allowed</code>	
<code>functionality_unimplemented</code>	(debugging version)
<code>clip_to_frame_shape_unimplemented</code>	(debugging version)

**Warnings**

<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)
<code>face_override_style_font_must_match_style</code>	

**Notices (debugging version)**

<code>shape_already_in_primitive_form</code>
--

## SEE ALSO

For examples using this function, see “Converting a Shape to Primitive Form” beginning on page 4-38.

For more information about the primitive form of shapes and for examples of functions that use shapes in their primitive form, see Chapter 3, “Geometric Styles,” in this book.

To eliminate unnecessary geometric points, use the `GXReduceShape` function, described on page 4-74. To simplify a shape’s contours, use the `GXSimplifyShape` function, described on page 4-76.

## Finding Geometric Information About Shapes

---

The functions described in this section calculate geometric information about a shape.

The `GXGetShapeLength` function calculates the length of a particular contour or of all contours in a shape.

The `GXShapeLengthToPoint` function determines the point that falls at a specified distance along a particular contour or along all combined contours of a shape.

The `GXGetShapeCenter` function determines the center point of a particular contour or of all combined contours of a shape's geometry.

The `GXGetShapeArea` function calculates the area covered by a particular contour or by all combined contours of a shape's geometry.

You can also use the `GXGetShapeBounds` function, described on page 4-90, to find geometric information about a shape—in this case, the shape's bounding rectangle.

### GXGetShapeLength

---

You can use the `GXGetShapeLength` function to determine the length of a particular contour or of all contours of a shape.

```
gxWide *GXGetShapeLength(gxShape source, long index,
                          gxWide *length);
```

`source`      A reference to the shape containing the contour.

`index`        The index of the contour you want to measure. You may specify a value of 0 for this parameter to measure all contours.

`length`       A pointer to a `gxWide` value. On return, this value indicates the length of the indicated contour.

*function result*   The length of the indicated contour.

#### DESCRIPTION

The `GXGetShapeLength` function returns as the function result the length of the perimeter of a particular contour of a shape. This function calculates the length of the contour as defined in the shape's geometry; it does not consider transformations to the shape made by the shape's transform.

## Geometric Operations

For empty and full shapes, this function posts the warning `shape_does_not_have_length`. For point shapes, it returns zero. For solid polygon and path shapes, this function calculates the length as if the shape had the closed-frame shape fill.

If you provide a target shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts the warning <code>shape_does_not_have_length</code>
picture	Returns the length of the specified picture item or the sum of the length of all picture items
text	Posts the warning <code>shape_does_not_have_length</code>
glyph	Posts the warning <code>shape_does_not_have_length</code>
layout	Posts the warning <code>shape_does_not_have_length</code>

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>contour_is_less_than_zero</code>	(debugging version)

**Warnings**

<code>contour_out_of_range</code>	
<code>shape_does_not_have_length</code>	(debugging version)
<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)

## SEE ALSO

For an example using this function, see “Finding the Length of a Contour” beginning on page 4-42.

For information about the contours of the various geometric shapes, see Chapter 2, “Geometric Shapes,” in this book.

## GXShapeLengthToPoint

---

You can use the `GXShapeLengthToPoint` function to determine the point that falls at a certain distance along a contour of a shape.

```
gxPoint *GXShapeLengthToPoint(gxShape target, long index,
                               Fixed length, gxPoint *location,
                               gxPoint *tangent);
```

<code>target</code>	A reference to the shape containing the contour you want to examine.
<code>index</code>	The number of the contour within the shape. You may specify a value of 0 for this parameter to indicate that the function should start measuring at the beginning of the first contour and continue through all contours.
<code>length</code>	The distance along the specified contour.
<code>location</code>	A pointer to a <code>gxPoint</code> structure. On return, this structure contains the point that lies along the contour at the specified distance.
<code>tangent</code>	A pointer to a <code>gxPoint</code> structure. On return, this structure contains a point that specifies a tangent vector representing the slope of the contour at the specified distance.
<i>function result</i>	The point that lies along the contour at the specified distance. (This value is the same as the value returned in the <code>location</code> parameter.)

### DESCRIPTION

The `GXShapeLengthToPoint` function returns the location of the point that lies at the distance specified by the `length` parameter along the contour specified by the `index` parameter of the target shape.

If you provide a pointer for the `tangent` parameter that is not `nil`, this function returns the slope of the specified contour at that point, in the form of a tangent vector. (The tangent vector implicitly starts at point (0.0, 0.0) and ends at the point indicated by the returned `tangent` parameter.)

This function measures the contour length as defined in the shape's geometry; it does not consider transformations to the shape made by the shape's transform.

## Geometric Operations

If the target shape has the `noFill` shape fill, this function posts the error `shape_fill_not_allowed`.

If the target shape is an empty shape or a full shape, this function posts the warning `shape_does_not_have_length`.

If you provide a target shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
<code>bitmap</code>	Posts the warning <code>shape_does_not_have_length</code>
<code>picture</code>	Posts the warning <code>shape_does_not_have_length</code>
<code>text</code>	Posts the warning <code>shape_does_not_have_length</code>
<code>glyph</code>	Posts the warning <code>shape_does_not_have_length</code>
<code>layout</code>	Posts the warning <code>shape_does_not_have_length</code>

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>length_is_less_than_zero</code>	(debugging version)
<code>parameter_is_nil</code>	(debugging version)
<code>shape_fill_not_allowed</code>	(debugging version)

**Warnings**

<code>contour_out_of_range</code>	
<code>length_out_of_range</code>	
<code>shape_does_not_have_length</code>	(debugging version)
<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)

## SEE ALSO

For an example using this function, see “Finding the Point at a Certain Distance Along a Contour” beginning on page 4-42.

For information about the contours of the various geometric shapes, see Chapter 2, “Geometric Shapes,” in this book.

To measure the length of a contour, use the `GXGetShapeLength` function, described on page 4-83.



## GXGetShapeCenter

---

You can use the `GXGetShapeCenter` function to determine the center of a specified contour of a shape.

```
gxPoint *GXGetShapeCenter(gxShape source, long index,
                          gxPoint *center);
```

<code>source</code>	A reference to the shape containing the contour whose center you want to find.
<code>index</code>	The number of the contour whose center you want to find. You may specify a value of 0 to indicate you want to find the center of the entire shape.
<code>center</code>	A pointer to a <code>gxPoint</code> structure. On return, this structure contains the point that falls at the center of the specified contour.

*function result* The point that falls at the center of the specified contour. (This value is the same as the value returned in the `center` parameter.)

### DESCRIPTION

The `GXGetShapeCenter` function determines the point that falls at the center of the contour specified by the `index` parameter of the shape specified by the `source` parameter. If you specify a value of 0 for the `index` parameter, this function finds the center point of the entire source shape.

The center point of a shape is not merely the center of the shape's bounding rectangle; rather it is the "center of gravity" of a shape. QuickDraw GX guarantees the center point of a shape does not change even if the shape is rotated.

This function finds the center of a shape (or of a particular contour) as defined by the source shape's geometry; it does not consider shape fill or transformations to the shape made by the shape's transform. For point shapes, this function returns a copy of the point's geometry. For line and rectangle shapes, this function returns the midpoint of the geometry. For empty and full shapes, this function posts the error `shape_does_not_have_length`.

If you provide a target shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Returns midpoint of bounding rectangle
picture	Posts the error <code>illegal_type_for_shape</code>
text	Returns midpoint of the bounding rectangle of the specified glyph
glyph	Returns midpoint of the bounding rectangle of the specified glyph
layout	Converts to glyph shape and returns midpoint of the bounding rectangle of the specified glyph

#### ERRORS, WARNINGS, AND NOTICES

##### Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>contour_less_than_zero</code>	(debugging version)
<code>graphic_type_does_not_contain_points</code>	(debugging version)

##### Warnings

<code>contour_out_of_range</code>	
<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)

#### SEE ALSO

For examples using this function, see “Finding the Bounding Rectangle and Center Point of a Shape” beginning on page 4-43.

To find the bounding rectangle of a shape or a contour of a shape, use the `GXGetShapeBounds` function described on page 4-90.

## GXGetShapeArea

---

You can use the `GXGetShapeArea` function to determine the area covered by a specific contour of a shape's geometry.

```
gxWide *GXGetShapeArea(gxShape source, long index, gxWide *area);
```

<code>source</code>	A reference to the shape containing the contour whose area you want to determine.
---------------------	---

## Geometric Operations

<i>index</i>	The number of the contour whose area you want to determine. You may specify a value of 0 for this parameter to indicate you want to determine the area of the entire shape.
<i>area</i>	A pointer to a <code>gxWide</code> value. On return, this value indicates the area covered by the contour.
<i>function result</i>	The area covered by the contour. (This value is the same as the value returned in the <code>area</code> parameter.)

## DESCRIPTION

The `GXGetShapeArea` function returns the area covered by the contour specified by the `index` parameter of the shape indicated by the `source` parameter. This function considers only the geometry of the source shape—it does not consider the shape fill of the shape. The same geometry returns the same area whether the shape has one of the framed fills, an even-odd fill, a winding fill, or one of the inverse fills.

Some shapes have overlapping contours with the same contour direction. (When drawing these shapes, QuickDraw GX fills these overlapping areas if the shape has a winding fill and does not fill these areas if the shape has an even-odd fill.) The `GXGetShapeArea` function counts these overlapping areas twice. To correct this calculation, call the `GXSimplifyShape` function before calling the `GXGetShapeArea` function:

- n For shapes with a winding shape fill, the `GXSimplifyShape` function eliminates the inner contour and, therefore, the `GXGetShapeArea` function counts the overlapping area only once.
- n For shapes with an even-odd shape fill, the `GXSimplifyShape` function reverses the contour direction of the internal contour, and therefore the `GXGetShapeArea` function does not count the overlapping area at all.

This function measures the shape area as defined in the shape's geometry; it does not consider transformations to the shape made by the shape's transform.

For empty shapes, point shapes, and line shapes, this function posts the error `shape_does_not_have_area`. For full shapes, it posts the error `illegal_type_for_shape`.

If you provide a target shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Returns the bitmap height multiplied by the bitmap width
picture	Returns the sum of the areas of the picture items
text	Converts to path shape and finds area
glyph	Converts to path shape and finds area
layout	Converts to path shape and finds area

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`  
`shape_is_nil`  
`parameter_is_nil`  
`shape_does_not_have_area` (debugging version)

**Warnings**

`index_out_of_range`  
`contour_out_of_range`

**SEE ALSO**

For examples using this function, see “Finding the Area of a Shape” beginning on page 4-45.

For a discussion of shape fills, see Chapter 2, “Geometric Shapes,” in this book

To simplify a shape before measuring its area, use the `GXSimplifyShape` function, described on page 4-76.

**Getting and Setting Shape Bounds**

---

Every shape has a bounding rectangle—the smallest rectangle that contains the shape. The functions in this section allow you to determine and alter a shape’s bounding rectangle.

The `GXGetShapeBounds` function finds the bounding rectangle of a shape, or of a specified contour of a shape.

The `GXSetShapeBounds` function allows you to alter a shape’s bounding rectangle (and thereby move and resize the shape).

**GXGetShapeBounds**

---

You can use the `GXGetShapeBounds` function to determine the bounding rectangle of a shape or of a specified contour of a shape.

```
gxRectangle *GXGetShapeBounds(gxShape source, long index,
                              gxRectangle *bounds);
```

<code>source</code>	A reference to the shape containing the contour whose bounding rectangle you want to find.
<code>index</code>	The number of the contour whose bounding rectangle you want to find. You may specify a value of 0 to indicate you want to find the bounding rectangle of the entire shape.

## Geometric Operations

**bounds** A pointer to a `gxRectangle` structure. On return, this structure contains the bounding rectangle of the specified contour.

**function result** The bounding rectangle of the specified contour. (This value is the same as the value returned in the `bounds` parameter.)

## DESCRIPTION

The `GXGetShapeBounds` function determines the bounding rectangle of the contour specified by the `index` parameter of the shape specified by the `source` parameter. If you specify a value of 0 for the `index` parameter, this function finds the bounding rectangle of the entire source shape.

The bounding rectangle of a shape (or of a contour of a shape) is the smallest rectangle that contains the geometry of the shape (or of the contour).

This function finds the bounding rectangle of the source shape (or a contour of the source shape) as defined by the source shape's geometry; it does not consider shape fill or transformations to the shape made by the shape's transform.

For empty shapes and full shapes, this function posts the warning `shape_passed_has_no_bounds`. For full shapes, it returns an infinitely large rectangle; for empty shapes, it returns the inverse rectangle.

If you provide a target shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Returns the bounding rectangle
picture	Returns the bounding rectangle for the entire picture
text	Returns bounding rectangle of specified glyphs
glyph	Returns bounding rectangle of specified glyphs
layout	Posts the error <code>functionality_unimplemented</code> if the <code>index</code> parameter is not zero; returns bounding rectangle of glyphs otherwise

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>contour_is_less_than_zero</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>functionality_unimplemented</code>	(debugging version)

**Warnings**

<code>contour_out_of_range</code>	
<code>shape_passed_has_no_bounds</code>	(debugging version)
<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)

## SEE ALSO

For a discussion of rectangles and bounding rectangles, see Chapter 2, “Geometric Shapes,” in this book.

To find the center of a shape or a contour of a shape, use the `GXGetShapeCenter` function, which is described on page 4-87.

To change the bounding rectangle of a shape, use the `GXSetShapeBounds` function, described in the next section.

## GXSetShapeBounds

---

You can use the `GXSetShapeBounds` function to change a shape’s bounding rectangle, thereby moving and resizing the shape.

```
void GXSetShapeBounds(gxShape target,
                     const gxRectangle *newBounds);
```

**source**        A reference to the shape whose bounding rectangle you want to change.

**newBounds**    The new bounding rectangle.

## DESCRIPTION

The `GXSetShapeBounds` function changes the bounding rectangle of the shape specified by the `source` parameter to be the rectangle specified by the `newBounds` parameter.

How this function changes the bounding rectangle is determined by the source shape’s `gxMapTransformShape` shape attribute:

- n If the `gxMapTransformShape` shape attribute is not set, the function changes the geometry of the source shape to fit the new bounding rectangle.
- n If the `gxMapTransformShape` shape attribute is set, the function does not alter the shape’s geometry directly; instead, it changes the mapping of the shape’s transform object to scale the shape to fit in the new bounding rectangle.

By changing a shape’s bounding rectangle, you can move the shape as well as scale it in the horizontal and vertical dimensions.

For empty and full shapes, this function does nothing.

If you provide a point shape as the target shape and a new bounding rectangle that has height or width, this function posts the warning `scale_shape_out_of_range`.

If you provide a target shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Calls the <code>GXMapShape</code> function
picture	Calls the <code>GXMapShape</code> function
text	Converts to a path shape if necessary (when the ratio between the height of the new bounding rectangle and the height of the original bounding rectangle is not the same as the ratio between the width of the new bounding rectangle and the width of the original bounding rectangle)
glyph	Converts to a path shape if necessary
layout	Converts to a path shape if necessary

#### ERRORS, WARNINGS, AND NOTICES

##### Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>shape_access_not_allowed</code>	(debugging version)
<code>functionality_unimplemented</code>	(debugging version)

##### Warnings

<code>scale_shape_out_of_range</code>	
<code>character_substitution_took_place</code>	
<code>font_substitution_took_place</code>	
<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)
<code>shape_passed_has_no_bounds</code>	(debugging version)

#### SEE ALSO

For examples of this function, see “Setting a Shape’s Bounding Rectangle” beginning on page 4-47.

For a discussion of rectangles and bounding rectangles, see Chapter 2, “Geometric Shapes,” in this book.

For a discussion of the `gxMapTransformShape` `shape` attribute, see the chapters “Shape Objects” and “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To determine the bounding rectangle of a shape, use the `GXGetShapeBounds` function, described on page 4-90.

## Insetting Shapes

---

The `GXInsetShape` function, described in this section, provides a way to inset or outset the contours of a shape a specified distance from the original contours.

### GXInsetShape

---

You can use the `GXInsetShape` procedure to inset a shape's geometry.

```
void GXInsetShape(gxShape target, Fixed inset);
```

`source`        A reference to the shape whose geometry you want to inset.

`inset`         The distance to inset the geometry of the shape.

#### DESCRIPTION

The `GXInsetShape` function insets the geometry of the shape specified by the `target` parameter by the distance specified in the `inset` parameter. The on-curve geometric points of the resulting geometry are the specified distance inside the contour of the original geometry.

You can specify a positive or negative value for the `inset` parameter: positive values move the geometry to the inside of the original geometry; negative values move it outside the original geometry.

QuickDraw GX uses the direction of a contour to define which side is the inside of a contour: the inside is the side to the right of the contour. As a result, insetting clockwise contours by a positive amount makes them smaller while insetting counterclockwise contours by a positive amount makes them larger.

You can override this behavior by setting the `gxAutoInsetStyle` style attribute. If you set this style attribute for a shape, QuickDraw GX finds the true inside of the contour, regardless of its contour direction. With this attribute set, insetting a contour by a positive amount makes it smaller, whether it has a clockwise direction or a counterclockwise direction.

If the target shape has the `noFill` shape fill, this function posts the error `shape_fill_not_allowed`.

For empty, full, and point shapes, this function posts the error `graphic_type_cannot_be_inset`. Line shapes and rectangle shapes are converted to polygon shapes; curve shapes are converted to path shapes.



If you provide a target shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts the error <code>graphic_type_cannot_be_inset</code>
picture	Posts the error <code>graphic_type_cannot_be_inset</code>
text	Posts the error <code>graphic_type_cannot_be_inset</code>
glyph	Posts the error <code>graphic_type_cannot_be_inset</code>
layout	Posts the error <code>graphic_type_cannot_be_inset</code>

## ERRORS, WARNINGS, AND NOTICES

### Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>graphic_type_cannot_be_inset</code>	(debugging version)
<code>shape_fill_not_allowed</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)

### Warnings

<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)
--	---------------------

### Notices (debugging version)

<code>geometry_unaffected</code>	(debugging version)
----------------------------------	---------------------

## SEE ALSO

For examples using this function, see “Insetting Shapes” beginning on page 4-50.

For a discussion of contours and contour direction, see Chapter 2, “Geometric Shapes,” in this book.

For a discussion of the `gxAutoInsetStyle` style attribute, see Chapter 3, “Geometric Styles,” in this book.

To change the bounding rectangle of a shape, use the `GXSetShapeBounds` function, described on page 4-92.

## Determining Whether Two Areas Touch

---

The functions described in this section determine if two areas touch.

The `GXTouchesRectanglePoint` function determines whether the area covered by a rectangle touches a point.

The `GXTouchesBoundsShape` function determines whether the area covered by a rectangle touches a shape.

The `GXTouchesShape` function determines whether the area covered by one shape touches the area covered by another.

The `GXIntersectShape` function, which is described on page 4-107 in the section “Performing Geometric Arithmetic With Shapes,” determines not only if two shapes intersect but also what their intersection is.

## GXTouchesRectanglePoint

---

You can use the `GXTouchesRectanglePoint` function to determine if a point lies within or on the edge of a rectangle.

```
gxBoolean GXTouchesRectanglePoint(const gxRectangle *target,
                                   const gxPoint *test);
```

`target`      A pointer to the rectangle to test as the container.

`test`        A pointer to the point to test for inclusion.

*function result* A Boolean value indicating whether the point touches the rectangle.

### DESCRIPTION

The `GXTouchesRectanglePoint` function returns `true` as its function result if the point specified by the `test` parameter lies within or on the edge of the rectangle specified by the `target` parameter, and returns `false` otherwise.

Notice that the parameters to this function are not shapes; they are pointers to a `gxPoint` or to a `gxRectangle` structure.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`parameter_is_nil`

### SEE ALSO

For a discussion of the `gxPoint` and `gxRectangle` data structures, see Chapter 2, “Geometric Shapes,” in this book.

To determine if a rectangle touches a shape, use the `GXTouchesBoundsShape` function, described in the next section.

To determine if a rectangle contains a shape, use the `GXContainsBoundsShape` function, described on page 4-101.

## GXTouchesBoundsShape

---

You can use the `GXTouchesBoundsShape` function to determine if a rectangle and a shape touch.

```
gxBoolean GXTouchesBoundsShape(const gxRectangle *target,
                                gxShape test);
```

`target`      A pointer to the rectangle to test to determine if it touches a shape.

`test`        A reference to the shape to test to determine if it touches the rectangle.

*function result* A Boolean value indicating whether the shape touches the rectangle.

### DESCRIPTION

The `GXTouchesBoundsShape` function returns `true` as its function result if the rectangle specified by the `target` parameter touches the shape specified by the `test` parameter—even if they share only an edge or a point—and returns `false` otherwise.

This function considers the shape fill, the style modifications, and the transform mapping of the test shape. Only areas that are drawn are considered when determining touching.

If you provide a test shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Compares bounding rectangle of bitmap
picture	Posts the error <code>illegal_type_for_shape</code>
text	Converts to path shape
glyph	Converts to path shape
layout	Converts to path shape

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>illegal_type_for_shape</code>	(debugging version)
<code>shape_may_not_be_a_picture</code>	(debugging version)

## SEE ALSO

For an example using this function, see “Determining Whether Two Shapes Touch” beginning on page 4-53.

For a discussion of shape fills, see Chapter 2, “Geometric Shapes,” in this book.

For a discussion of style modifications, see Chapter 3, “Geometric Styles,” in this book.

For a discussion of transform mappings, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To determine if a rectangle touches a point, use the `GXTouchesRectanglePoint` function, described on page 4-96.

To determine if a rectangle contains a shape, use the `GXContainsBoundsShape` function, described on page 4-101.

To determine if two shapes touch, use the `GXTouchesShape` function, described in the next section.

**GXTouchesShape**

---

You can use the `GXTouchesShape` function to determine if two shapes touch.

```
gxBoolean GXTouchesShape(gxShape target, gxShape test);
```

`target`      A reference to one shape to test to determine if it touches another.

`test`        A reference to the other shape to test.

*function result*   A Boolean value indicating whether the shapes intersect.

**DESCRIPTION**

The `GXTouchesShape` function returns `true` as its function result if the shape specified by the `target` parameter touches the shape specified by the `test` parameter—even if they share only an edge or a point—and returns `false` otherwise.

This function considers the shape fill, the style modifications, and the transform mapping of the target and test shapes. Only areas that are drawn are considered when determining touching.

For example, if the target shape has an even-odd fill and contains an overlapping contour, then the shape has an internal area that is not drawn. If the test shape lies entirely within this area, the `GXTouchesShape` function returns `false`.

As another example, if the test shape lies entirely within the target shape, but the target shape has an inverse shape fill, the `GXTouchesShape` function returns `false`.

If you provide a target or test shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
<code>bitmap</code>	Compares bounding rectangle of bitmap
<code>picture</code>	Posts the error <code>illegal_type_for_shape</code>
<code>text</code>	Converts to path shape
<code>glyph</code>	Converts to path shape
<code>layout</code>	Converts to path shape

**ERRORS, WARNINGS, AND NOTICES****Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>illegal_type_for_shape</code>	(debugging version)
<code>shape_may_not_be_a_picture</code>	(debugging version)

**SEE ALSO**

For examples using this function, see “Determining Whether Two Shapes Touch” beginning on page 4-53.

For a discussion of shape fills, see Chapter 2, “Geometric Shapes,” in this book.

For a discussion of style modifications, see Chapter 3, “Geometric Styles,” in this book.

For a discussion of transform mappings, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To determine if a rectangle touches a shape, use the `GXTouchesBoundsShape` function, described on page 4-97.

To determine if a shape contains another shape, use the `GXContainsShape` function, described on page 4-103.

## Determining Whether One Shape Contains Another

---

The functions described in this section determine if one area contains another.

The `GXContainsRectangle` function determines whether the area covered by one rectangle contains the area covered by another.

The `GXContainsBoundsShape` function determines whether the area covered by a rectangle contains the area covered by a shape.

The `GXContainsShape` function determines whether the area covered by one shape contains the area covered by another.

### `GXContainsRectangle`

---

You can use the `GXContainsRectangle` function to determine if one rectangle contains another.

```
gxBoolean GXContainsRectangle(const gxRectangle *container,
                             const gxRectangle *test);
```

`container`    A pointer to the rectangle to test as the container.

`test`        A pointer to the rectangle to test for inclusion.

*function result*    A Boolean value indicating whether the container rectangle contains the test rectangle.

#### DESCRIPTION

The `GXContainsRectangle` function returns `true` as its function result if the rectangle specified by the `test` parameter lies within the rectangle specified by the `target` parameter, and `false` otherwise.

This function may return `true` even if the container and test rectangles share one or more edges. This function returns `true` when the container and test rectangles are defined by the same coordinates.

Notice that the parameters to this function are not shapes; they are pointers to `gxRectangle` structures.

## ERRORS, WARNINGS, AND NOTICES

**Errors**`parameter_is_nil`

## SEE ALSO

For a discussion of the `gxRectangle` data structure, see Chapter 2, “Geometric Shapes,” in this book.

To determine if a rectangle touches a point, use the `GXTouchesRectanglePoint` function, described on page 4-96.

To determine if a rectangle contains a shape, use the `GXContainsBoundsShape` function, described in the next section.

**GXContainsBoundsShape**

---

You can use the `GXContainsBoundsShape` function to determine if a rectangle contains a shape or a particular contour of a shape.

```
gxBoolean GXContainsBoundsShape(const gxRectangle *container,
                                gxShape test, long index);
```

`container`    A pointer to the rectangle to test as the container.

`test`        A reference to the shape containing the contour to test for inclusion.

`index`       The number of the contour to test for inclusion. You may specify a value of 0 to indicate you want to test the entire shape for inclusion.

*function result*    A Boolean value indicating whether the container rectangle contains the specified contour of the test shape.

**DESCRIPTION**

The `GXContainsBoundsShape` function returns `true` as its function result if the rectangle specified by the `container` parameter contains the contour indicated by the `index` parameter of the shape specified by the `test` parameter and returns `false` otherwise.

This function may return `true` even if the container rectangle and the indicated contour of the test shape share one or more edges.

## Geometric Operations

This function considers the shape fill, the style modifications, and the transform mapping of the test shape. Only areas that are drawn are considered when determining whether the container rectangle contains the specified contour.

If you provide a test shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Compares bounding rectangle of bitmap
picture	Compares bounding rectangle of entire picture
text	Converts to path shape
glyph	Converts to path shape
layout	Converts to path shape

## ERRORS, WARNINGS, AND NOTICES

**Errors**

out_of_memory	
shape_is_nil	
parameter_is_nil	(debugging version)
illegal_type_for_shape	(debugging version)
shape_operator_may_not_be_a_picture	(debugging version)

## SEE ALSO

For a discussion of shape fills, see Chapter 2, “Geometric Shapes,” in this book.

For a discussion of style modifications, see Chapter 3, “Geometric Styles,” in this book.

For a discussion of transform mappings, see the chapter “Transform Objects” of *Inside Macintosh: QuickDraw GX Objects*.

To determine if a rectangle touches a shape, use the `GXTouchesBoundsShape` function, described on page 4-97.

To determine if a shape contains another shape, use the `GXContainsShape` function, described in the next section.



## GXContainsShape

---

You can use the `GXContainsShape` function to determine if the area covered by a shape contains the area covered by another shape.

```
gxBoolean GXContainsShape(gxShape container, gxShape test);
```

`container`    A reference to the shape to test as the container.

`test`            A reference to the shape to test for inclusion.

*function result*    A Boolean value indicating whether the container shape contains the test shape.

### DESCRIPTION

The `GXContainsShape` function returns `true` as its function result if the shape specified by the `container` parameter contains the shape specified by the `test` parameter, and returns `false` otherwise.

This function may return `true` even if the container shape and the test shape share one or more edges; it returns `true` if they are the same shape.

This function considers the shape fill, the style modifications, and the transform mapping of the container and test shapes. Only areas that are drawn are considered when determining whether the container shape contains the test shape.

The container shape must have one of the solid shape fills (even-odd, winding, inverse even-odd, or inverse winding). The test shape may have any shape fill.

If the test shape has a framed shape fill, this function returns `true` if the frame lies entirely within the area of the container shape, or along the edges of the container shape. As a result, a solid shape contains its own frame.

If you provide a test shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Compares bounding rectangle of bitmap
picture	Posts the error <code>illegal_type_for_shape</code>
text	Converts to path shape
glyph	Converts to path shape
layout	Converts to path shape

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>illegal_type_for_shape</code>	(debugging version)
<code>shape_operator_may_not_be_a_picture</code>	(debugging version)

## SEE ALSO

For examples using this function, see “Determining Whether One Shape Contains Another” beginning on page 4-58.

For a discussion of shape fills, see Chapter 2, “Geometric Shapes,” in this book.

For a discussion of style modifications, see Chapter 3, “Geometric Styles,” in this book.

For a discussion of transform mappings, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To determine if a rectangle contains a shape, use the `GXContainsBoundsShape` function, described on page 4-101.

To determine if one shape touches another, use the `GXTouchesShape` function, described on page 4-98.

## Performing Geometric Arithmetic With Shapes

---

QuickDraw GX provides six arithmetic operations you can apply to geometric shapes: intersection, union, difference, reverse difference, exclusion, and inversion.

The `GXIntersectRectangle` and `GXUnionRectangle` perform the intersection and union operations on rectangle structures.

The other functions described in this section perform the arithmetic operations on shapes:

- n The `GXIntersectShape` function finds the area common to the shapes.
- n The `GXUnionShape` function finds the smallest area that contains both the shapes.
- n The `GXDifferenceShape` function finds the area covered by the first shape that is not covered by the second shape.
- n The `GXReverseDifferenceShape` function finds the area covered by the second shape that is not covered by the first shape.
- n The `GXExcludeShape` function finds the area covered by one shape or the other, but not by both.
- n The `GXInvertShape` function finds the area not covered by a shape.

## GXIntersectRectangle

---

You can use the `GXIntersectRectangle` function to find the intersection of two rectangles.

```
gxBoolean GXIntersectRectangle(gxRectangle *target,
                               const gxRectangle *source,
                               const gxRectangle *operand);
```

**target**      A pointer to a `gxRectangle` structure. On return, the intersection of the source and operand rectangles. You may specify the value `nil` for this parameter if you do not want the intersection to be calculated. Depending on the result of the intersection operation, this pointer may point to the source or operand rectangle.

**source**      A pointer to one of the rectangles to intersect.

**operand**     A pointer to the other rectangle to intersect.

*function result* A Boolean value indicating whether the rectangles intersect.

### DESCRIPTION

The `GXIntersectRectangle` function returns `true` as its function result if the source rectangle and the operand rectangle intersect, and returns `false` otherwise.

If you provide a pointer for the `target` parameter that is not `nil`, this function returns the intersection of the source and operand rectangles in the `gxRectangle` structure pointed to by the `target` parameter.

If the source rectangle and the operand rectangle do not intersect or share only one edge, this function returns `false` and does not affect the target rectangle.

You may specify the source rectangle or the operand rectangle as the target rectangle. In this case, the function calculates the intersection of the original rectangles and then places the calculated intersection into the source or operand rectangle, as specified.

Notice that the parameters to this function are not shapes; they are pointers to `gxRectangle` data structures.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

`parameter_is_nil`

**SEE ALSO**

For a discussion of the `gxRectangle` data structure, see Chapter 2, “Geometric Shapes.”

For a discussion of geometric arithmetic, see “Geometric Arithmetic” beginning on page 4-21.

To find the intersection of two shapes, use the `GXIntersectShape` function, described on page 4-107.

To find the union of two rectangles, use the `GXUnionRectangle` function, described in the next section.

## **GXUnionRectangle**

---

You can use the `GXUnionRectangle` function to find the smallest rectangle that contains two other rectangles.

```
gxRectangle *GXUnionRectangle(gxRectangle *target,
                              const gxRectangle *source,
                              const gxRectangle *operand);
```

**target**      A pointer to a `gxRectangle` structure. On return, the smallest rectangle containing both the source and operand rectangles.

**source**      A pointer to one of the rectangles to combine.

**operand**     A pointer to the other rectangle to combine.

*function result*   The smallest rectangle containing both the source and operand rectangles. (This rectangle is the same as the rectangle returned in the `target` parameter.)

**DESCRIPTION**

The `GXUnionRectangle` function calculates the smallest rectangle containing both the source rectangle and the operand rectangle and stores the results in `target` parameter. This function also returns the calculated rectangle as its function result.

You may specify the source rectangle or the operand rectangle as the target rectangle. In this case, the function calculates the smallest rectangle containing both of the original rectangles and then places the calculated rectangle into the source or operand rectangle, as specified.

Notice that the parameters to this function are not shapes, but pointers to the `gxRectangle` data structures.

## ERRORS, WARNINGS, AND NOTICES

**Errors**`parameter_is_nil`

## SEE ALSO

For a discussion of the `gxRectangle` data structure, see Chapter 2, “Geometric Shapes,” in this book.

For a discussion of geometric arithmetic, see “Geometric Arithmetic” beginning on page 4-21.

To find the intersection of two rectangles, use the `GXIntersectRectangle` function, described in this previous section.

To find the union of two shapes, use the `GXUnionShape` function, described on page 4-109.

**GXIntersectShape**

---

You can use the `GXIntersectShape` function to find the intersection of two shapes.

```
void GXIntersectShape(gxShape target, gxShape operand);
```

**target**      On input, a reference to one of the shapes to intersect. On output, a reference to the intersection of the input target shape and the operand shape.

**operand**     A reference to the other shape to intersect.

## DESCRIPTION

The `GXIntersectShape` function finds the intersection of the target shape and the operand shape, reduces and simplifies the result, and stores it in the target shape. If the original target shape and the operand shape do not intersect, the resulting target shape is an empty shape.

If the target shape and the operand shape share only an edge, and if both have a solid fill, the resulting target shape is an empty shape. However, you can provide a framed target shape and a solid operand shape—the result being a framed shape.

This function considers the shape fill, the style modifications, and the transform mapping of the target and operand shapes. Only areas that are drawn are considered when determining intersection.

**Implementation Note**

Due to an implementation limit with QuickDraw GX version 1.0, you can find the intersection of two framed shapes only if the shapes are points, lines, or curves. [u](#)

## Geometric Operations

If you provide a target or operand shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts the error <code>shape_operator_may_not_be_a_bitmap</code>
picture	Posts the error <code>shape_operator_may_not_be_a_picture</code>
text	Converts to path shape if other parameter is not an empty shape or a full shape
glyph	Converts to path shape if other parameter is not an empty shape or a full shape
layout	Converts to path shape if other parameter is not an empty shape or a full shape

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>number_of_contours_exceeds_implementation_limit</code>	
<code>number_of_points_exceeds_implementation_limit</code>	
<code>size_of_path_exceeds_implementation_limit</code>	
<code>size_of_polygon_exceeds_implementation_limit</code>	
<code>fill_type_not_allowed</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)
<code>clip_to_frame_shape_unimplemented</code>	(debugging version)
<code>shape_operator_may_not_be_a_bitmap</code>	(debugging version)
<code>shape_operator_may_not_be_a_picture</code>	(debugging version)

**Warnings**

<code>character_substitution_took_place</code>	
<code>font_substitution_took_place</code>	
<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)

## SEE ALSO

For an example using this function, see “Performing Geometric Arithmetic With Shapes” beginning on page 4-60.

For a discussion of geometric arithmetic, see “Geometric Arithmetic” beginning on page 4-21.

For a discussion of shape fills, see Chapter 2, “Geometric Shapes,” in this book.

For a discussion of style modifications, see Chapter 3, “Geometric Styles,” in this book.

For a discussion of transform mappings, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To determine if two shapes touch, use the `GXTouchesShape` function, described on page 4-98.

To find the union of two shapes, use the `GXUnionShape` function, described in the next section.

## GXUnionShape

---

You can use the `GXUnionShape` function to find the union of two shapes.

```
void GXUnionShape(gxShape target, gxShape operand);
```

<code>target</code>	On input, a reference to one of the shapes to combine. On output, a reference to the union of the input target shape and the operand shape.
<code>operand</code>	A reference to the other shape to combine.

### DESCRIPTION

The `GXUnionShape` function finds the union of the target shape and the operand shape, reduces and simplifies the result, and stores it in the target shape.

This function considers the shape fill, the style modifications, and the transform mapping of the target and operand shape. Only areas that are drawn are considered when calculating the union.

The target shape and the operand shape must both have solid fills (even-odd, winding, inverse even-odd, or inverse winding) or both have framed fills (open-frame or closed-frame); one of each type of fill is not allowed.

If you provide a target or operand shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts the error <code>shape_operator_may_not_be_a_bitmap</code>
picture	Posts the error <code>shape_operator_may_not_be_a_picture</code>
text	Converts to path shape if other parameter is not an empty shape or a full shape
glyph	Converts to path shape if other parameter is not an empty shape or a full shape
layout	Converts to path shape if other parameter is not an empty shape or a full shape

## ERRORS, WARNINGS, AND NOTICES

**Errors**

out_of_memory	
shape_is_nil	
number_of_contours_exceeds_implementation_limit	
number_of_points_exceeds_implementation_limit	
size_of_path_exceeds_implementation_limit	
size_of_polygon_exceeds_implementation_limit	
fill_type_not_allowed	(debugging version)
shape_access_not_allowed	(debugging version)
clip_to_frame_shape_unimplemented	(debugging version)
shape_operator_may_not_be_a_bitmap	(debugging version)
shape_operator_may_not_be_a_picture	(debugging version)

**Warnings**

character_substitution_took_place	
font_substitution_took_place	
unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve	(debugging version)

## SEE ALSO

For examples using this function, see “Performing Geometric Arithmetic With Shapes” beginning on page 4-60.

For a discussion of geometric arithmetic, see “Geometric Arithmetic” beginning on page 4-21.

For a discussion of shape fills, see Chapter 2, “Geometric Shapes,” in this book.

For a discussion of style modifications, see Chapter 3, “Geometric Styles,” in this book.

For a discussion of transform mappings, see *Inside Macintosh: QuickDraw GX Objects*.

To find the intersection of two shapes, use the `GXIntersectShape` function, described on page 4-107.

**GXDifferenceShape**

---

You can use the `GXDifferenceShape` function to find the geometric difference between two shapes.

```
void GXDifferenceShape(gxShape target, gxShape operand);
```

target	On input, a reference to the shape to subtract from. On output, a reference to a shape describing the difference between the two shapes.
--------	--

operand	A reference to the shape to subtract.
---------	---------------------------------------



**DESCRIPTION**

The `GXDifferenceShape` function subtracts the operand shape from the target shape, reduces and simplifies the result, and stores it in the target shape.

The initial target shape does not have to contain the operand shape; the result of this function is the intersection of the target and operand shapes subtracted from the target shape.

This function considers the shape fill, the style modifications, and the transform mapping of the target and operand shapes: only areas that are drawn are considered when calculating the difference.

The operand shape cannot have one of the framed shape fills (open-frame or closed-frame). The target shape can have one of the framed fills; in this case, the resulting shape is the part of the frame that does not lie within the operand shape.

If you provide a target or operand shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts the error <code>shape_operator_may_not_be_a_bitmap</code>
picture	Posts the error <code>shape_operator_may_not_be_a_picture</code>
text	Converts to path shape if other parameter is not an empty shape or a full shape
glyph	Converts to path shape if other parameter is not an empty shape or a full shape
layout	Converts to path shape if other parameter is not an empty shape or a full shape

**ERRORS, WARNINGS, AND NOTICES****Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>number_of_contours_exceeds_implementation_limit</code>	
<code>number_of_points_exceeds_implementation_limit</code>	
<code>size_of_path_exceeds_implementation_limit</code>	
<code>size_of_polygon_exceeds_implementation_limit</code>	
<code>fill_type_not_allowed</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)
<code>clip_to_frame_shape_unimplemented</code>	(debugging version)
<code>shape_operator_may_not_be_a_bitmap</code>	(debugging version)
<code>shape_operator_may_not_be_a_picture</code>	(debugging version)

**Warnings**

<code>character_substitution_took_place</code>	
<code>font_substitution_took_place</code>	
<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)

**SEE ALSO**

For examples using this function, see “Performing Geometric Arithmetic With Shapes” beginning on page 4-60.

For a discussion of geometric arithmetic, see “Geometric Arithmetic” beginning on page 4-21.

For a discussion of shape fills, see Chapter 2, “Geometric Shapes,” in this book.

For a discussion of style modifications, see Chapter 3, “Geometric Styles,” in this book.

For a discussion of transform mappings, see *Inside Macintosh: QuickDraw GX Objects*.

For information about related routines, see the description of the `GXIntersectShape` function on page 4-107, the `GXUnionShape` function on page 4-109, and the `GXReverseDifferenceShape` function in the next section.

## **GXReverseDifferenceShape**

---

You can use the `GXReverseDifferenceShape` function to find the geometric difference between two shapes.

```
void GXReverseDifferenceShape(gxShape target, gxShape operand);
```

**target**        On input, a reference to the shape to subtract. On output, a reference to a shape describing the difference between the two shapes.

**operand**      A reference to the shape to subtract from.

**DESCRIPTION**

The `GXReverseDifferenceShape` function subtracts the target shape from the operand shape and stores the result in the target shape.

The initial operand shape does not have to contain the target shape; the result of this function is the intersection of the target and operand shapes subtracted from the operand shape.

This function considers the shape fill, the style modifications, and the transform mapping of the target and operand shapes. Only areas that are drawn are considered when calculating the difference.

Neither the target shape nor the operand shape have one of the framed fills (open-frame or closed-frame).

## Geometric Operations

If you provide a target or operand shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts the error <code>shape_operator_may_not_be_a_bitmap</code>
picture	Posts the error <code>shape_operator_may_not_be_a_picture</code>
text	Converts to path shape if other parameter is not an empty shape or a full shape
glyph	Converts to path shape if other parameter is not an empty shape or a full shape
layout	Converts to path shape if other parameter is not an empty shape or a full shape

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>number_of_contours_exceeds_implementation_limit</code>	
<code>number_of_points_exceeds_implementation_limit</code>	
<code>size_of_path_exceeds_implementation_limit</code>	
<code>size_of_polygon_exceeds_implementation_limit</code>	
<code>fill_type_not_allowed</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)
<code>clip_to_frame_shape_unimplemented</code>	(debugging version)
<code>shape_operator_may_not_be_a_bitmap</code>	(debugging version)
<code>shape_operator_may_not_be_a_picture</code>	(debugging version)

**Warnings**

<code>character_substitution_took_place</code>	
<code>font_substitution_took_place</code>	
<code>unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve</code>	(debugging version)

## SEE ALSO

For examples using this function, see “Performing Geometric Arithmetic With Shapes” beginning on page 4-60.

For a discussion of geometric arithmetic, see “Geometric Arithmetic” beginning on page 4-21.

For a discussion of shape fills, see Chapter 2, “Geometric Shapes,” in this book.

For a discussion of style modifications, see Chapter 3, “Geometric Styles,” in this book.

For a discussion of transform mappings, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

For information about related functions, see the description of the `GXIntersectShape` function on page 4-107, the `GXUnionShape` function on page 4-109, and the `GXDifferenceShape` function on page 4-110.

## GXExcludeShape

---

You can use the `GXExcludeShape` function to find the result of performing the exclusive-OR operation on two shapes.

```
void GXExcludeShape(gxShape target, gxShape operand);
```

target	On input, a reference to one of the shapes on which to perform the exclusive-OR operation. On output, a reference to a shape describing the exclusive-OR of the two shapes.
operand	A reference to the other shape on which to perform the exclusive-OR operation.

### DESCRIPTION

The `GXExcludeShape` function performs an exclusive-OR operation on the target and operand shapes, and stores the result in the target shape.

The exclusion of two shapes (the result of the exclusive-OR operation) is the area contained by the union of the two shapes less the area contained by the intersection of the two shapes.

This function considers the shape fill, the style modifications, and the transform mapping of the target and test shapes. Only areas that are drawn are considered when calculating the difference.

Neither the target shape nor the operand shape may have one of the framed fills (open-frame or closed-frame).

If you provide a target or operand shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts the error <code>shape_operator_may_not_be_a_bitmap</code>
picture	Posts the error <code>shape_operator_may_not_be_a_picture</code>
text	Converts to path shape if other parameter is not an empty shape or a full shape
glyph	Converts to path shape if other parameter is not an empty shape or a full shape
layout	Converts to path shape if other parameter is not an empty shape or a full shape

## ERRORS, WARNINGS, AND NOTICES

**Errors**

out_of_memory	
shape_is_nil	
number_of_contours_exceeds_implementation_limit	
number_of_points_exceeds_implementation_limit	
size_of_path_exceeds_implementation_limit	
size_of_polygon_exceeds_implementation_limit	
fill_type_not_allowed	(debugging version)
shape_access_not_allowed	(debugging version)
clip_to_frame_shape_unimplemented	(debugging version)
shape_operator_may_not_be_a_bitmap	(debugging version)
shape_operator_may_not_be_a_picture	(debugging version)

**Warnings**

character_substitution_took_place	
font_substitution_took_place	
unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve	(debugging version)

## SEE ALSO

For examples using this function, see “Performing Geometric Arithmetic With Shapes” beginning on page 4-60.

For a discussion of geometric arithmetic, see “Geometric Arithmetic” beginning on page 4-21.

For a discussion of shape fills, see Chapter 2, “Geometric Shapes,” in this book.

For a discussion of style modifications, see Chapter 3, “Geometric Styles,” in this book.

For a discussion of transform mappings, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

For information about related functions, see the description of the `GXIntersectShape` function on page 4-107, the `GXUnionShape` function on page 4-109, the `GXDifferenceShape` function on page 4-110, and the `GXReverseDifferenceShape` function on page 4-112.

## GXInvertShape

---

You can use the `GXInvertShape` function to invert a shape.

```
void GXInvertShape(gxShape target);
```

`target`      A reference to the shape to invert.

### DESCRIPTION

The `GXInvertShape` function inverts the target shape and stores the resulting shape in the target shape. Typically, this function changes the shape fill of the target shape. It also converts empty shapes to full shapes and full shapes to empty shapes.

If the target shape has one of the framed shape fills (open-frame or closed-frame), this function posts the error `shape_cannot_be_inverted`.

For empty shapes, this function converts the shape to a full shape; for full shapes, it converts to empty shapes.

If you provide a target or operand shape that is not one of the geometric shape types, this function performs the actions described in the following table:

Shape type	Action taken
bitmap	Posts the error <code>shape_cannot_be_inverted</code>
picture	Posts the error <code>shape_cannot_be_inverted</code>
text	Posts the error <code>shape_cannot_be_inverted</code>
glyph	Posts the error <code>shape_cannot_be_inverted</code>
layout	Posts the error <code>shape_cannot_be_inverted</code>

### ERRORS, WARNINGS, AND NOTICES

#### Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>shape_cannot_be_inverted</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)

### SEE ALSO

For an example using this function, see “Performing Geometric Arithmetic With Shapes” beginning on page 4-60.

For a discussion of geometric arithmetic, see “Geometric Arithmetic” beginning on page 4-21.

For a discussion of shape fills, see Chapter 2, “Geometric Shapes,” in this book.

## Summary of Geometric Operations

---

### Constants and Data Types

---

#### Contour Directions

```
enum gxContourDirections {
    gxCounterclockwiseDirection,
    gxClockwiseDirection
};

typedef long gxContourDirection;
```

### Functions

---

#### Determining and Reversing Contour Direction

```
gxContourDirection GXGetShapeDirection
                                (gxShape source, long contour);
void GXReverseShape              (gxShape target, long contour);
```

#### Breaking Shape Contours

```
void GXBreakShape                (gxShape target, long index);
```

#### Reducing and Simplifying Shapes

```
void GXReduceShape               (gxShape target, long contour);
void GXSimplifyShape             (gxShape target);
```

#### Incorporating Style Information Into Shape Geometries

```
void GXPrimitiveShape            (gxShape target);
```

#### Finding Geometric Information About Shapes

```
gxWide *GXGetShapeLength        (gxShape source, long index, gxWide *length);
gxPoint *GXShapeLengthToPoint   (gxShape target, long index, Fixed length,
                                gxPoint *location, gxPoint *tangent);
gxPoint *GXGetShapeCenter        (gxShape source, long index, gxPoint *center);
gxWide *GXGetShapeArea           (gxShape source, long index, gxWide *area);
```

**Getting and Setting Shape Bounds**

```

gxRectangle *GXGetShapeBounds
                                (gxShape source, long index,
                                 gxRectangle *bounds);

void GXSetShapeBounds           (gxShape target, const gxRectangle *newBounds);

```

**Insetting Shapes**

```

void GXInsetShape               (gxShape target, Fixed inset);

```

**Determining Whether Two Shapes Touch**

```

gxBoolean GXTouchesRectanglePoint
                                (const gxRectangle *target,
                                 const gxPoint *test);

gxBoolean GXTouchesBoundsShape
                                (const gxRectangle *target, gxShape test);

gxBoolean GXTouchesShape        (gxShape target, gxShape test);

```

**Determining Whether One Shape Contains Another**

```

gxBoolean GXContainsRectangle
                                (const gxRectangle *container,
                                 const gxRectangle *test);

gxBoolean GXContainsBoundsShape
                                (const gxRectangle *container, gxShape test,
                                 long index);

gxBoolean GXContainsShape        (gxShape container, gxShape test);

```

**Performing Geometric Arithmetic With Shapes**

```

gxBoolean GXIntersectRectangle
                                (gxRectangle *target, const gxRectangle *source,
                                 const gxRectangle *operand);

gxRectangle *GXUnionRectangle
                                (gxRectangle *target,
                                 const gxRectangle *source,
                                 const gxRectangle *operand);

void GXIntersectShape           (gxShape target, gxShape operand);
void GXUnionShape                (gxShape target, gxShape operand);
void GXDifferenceShape           (gxShape target, gxShape operand);
void GXReverseDifferenceShape
                                (gxShape target, gxShape operand);
void GXExcludeShape             (gxShape target, gxShape operand);
void GXInvertShape              (gxShape target);

```



# Bitmap Shapes

---

## Contents

About Bitmap Shapes	5-3
Bitmap Geometries	5-5
Bitmap Styles and Inks	5-8
Bitmap Transforms	5-10
Bitmaps and View Devices	5-12
Using Bitmap Shapes	5-14
Creating and Drawing Bitmaps	5-15
Creating Black-and-White Bitmaps	5-15
Creating Color Bitmaps	5-21
Dithering and Halftoning Bitmaps	5-30
Applying Transfer Modes to Bitmaps	5-32
Converting Other Types of Shapes to Bitmaps	5-34
Applying Transformations to Bitmaps	5-38
Mapping Bitmap Shapes	5-39
Clipping Bitmap Shapes	5-43
Creating Bitmaps With Disk-Based Pixel Images	5-44
Creating Bitmaps Offscreen	5-45
Editing Part of a Bitmap	5-53
Applying Functions Described Elsewhere to Bitmap Shapes	5-54
Functions That Post Errors or Warnings When Applied to Bitmap Shapes	5-55
Shape-Related Functions Applicable to Bitmap Shapes	5-56
Geometric Operations Applicable to Bitmap Shapes	5-58
Style-Related Functions Applicable to Bitmap Shapes	5-59
Ink-Related Functions Applicable to Bitmap Shapes	5-59
Transform-Related Functions Applicable to Bitmap Shapes	5-59
View-Related Functions Applicable to Bitmap Shapes	5-61
Bitmap Shapes Reference	5-61
Constants and Data Types	5-61
The Bitmap Geometry Structure	5-62

The Long Rectangle Structure	5-64
Constants For Bitmaps With Disk-Based Pixel Images	5-64
Bitmap Data Source Alias Structure	5-65
Functions	5-65
Creating Bitmaps	5-65
GXNewBitmap	5-66
Getting and Setting Bitmap Geometries	5-68
GXGetBitmap	5-68
GXSetBitmap	5-69
Editing Bitmaps	5-71
GXGetShapePixel	5-71
GXSetShapePixel	5-72
GXGetBitmapParts	5-74
GXSetBitmapParts	5-75
Drawing Bitmaps	5-76
GXDrawBitmap	5-77
Checking Bitmap Colors	5-79
GXCheckBitmapColor	5-79
Summary of Bitmap Shapes	5-81
Constants and Data Types	5-81
Functions	5-82

## Bitmap Shapes

This chapter describes bitmap shapes and the functions you use to manipulate them. It also discusses functions described in other chapters and shows how you can apply them to bitmap shapes.

Before you read this chapter, you should be familiar with the information in the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects*, and you will probably want to be familiar with much of the information discussed in the chapters “Color and Color-Related Objects,” “Transform Objects,” and “View-Related Objects,” also in that book.

This chapter introduces bitmap shapes, describes bitmap geometries, and then shows how to

- n define bitmap geometries
- n create bitmap shapes
- n draw bitmap shapes
- n manipulate the pixel image stored in a bitmap shape
- n apply transfer modes and transformations to bitmap shapes
- n draw other QuickDraw GX objects into a bitmap shape
- n create bitmap shapes with disk-based pixel images
- n replace a part of a bitmap shape’s pixel image

## About Bitmap Shapes

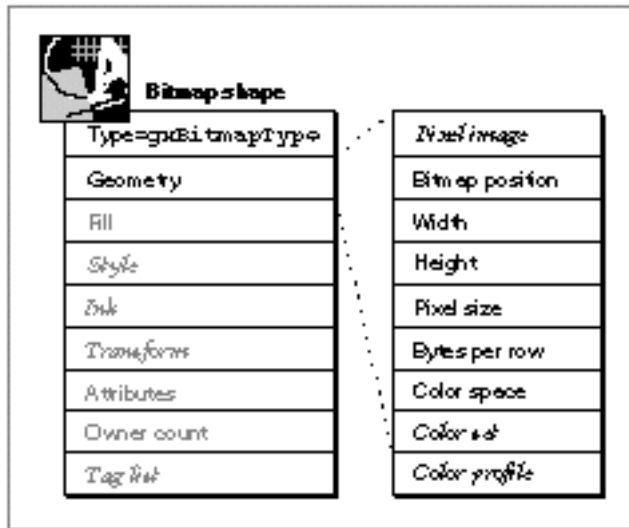
---

Like all shapes, a **bitmap shape** is represented in memory by a shape object, a style object, an ink object, and a transform object. A shape object representing a bitmap shape contains the same properties as a shape object representing a geometric or typographic shape: owner count, tag list, shape type, shape fill, geometry, and so on.

## Bitmap Shapes

Figure 5-1 shows a graphic representation of a bitmap shape and a bitmap geometry.

**Figure 5-1** A bitmap shape



Bitmap shapes make extensive use of their geometry property. In fact, most of the information useful to bitmap shapes is stored in their geometry—the values of the bitmap’s pixels, the dimensions of the bitmap, and the color information used by the bitmap.

Bitmap shapes don’t make much use of their shape fill property, and they use very little of their associated style object. In fact, the only information in a style object used by bitmap shapes are the style attributes that determine whether the upper-left corner of the bitmap should be constrained to an integer grid position.

Bitmap shapes don’t use the color property of their ink objects because they store their own color information in their geometries. However, they do use the transfer mode property of their ink objects.

Bitmap shapes do make full use of their transform objects. For example, you can scale, skew, rotate, and clip bitmap shapes. You can also hit-test bitmap shapes, but you cannot hit-test parts of a bitmap shape, as you can for other types of shapes. For more information about transform objects and hit-testing, see the chapter “Transform Objects” of *Inside Macintosh: QuickDraw GX Objects*.

The next few sections discuss bitmap geometries, bitmap styles, bitmap inks, and bitmap transforms.

## Bitmap Geometries

---

The geometry of a bitmap contains eight fields:

- n The **pixel image**—a pointer to a two-dimensional array of **pixel values**. Each pixel value represents the color of one pixel of the bitmap.
- n The **bitmap width**—the number of pixels in each row of the bitmap.
- n The **bitmap height**—the number of pixels in each column of the bitmap.
- n The **pixel size**—the number of bits required to represent the color information for each pixel of the bitmap.
- n The **bytes per row**—the number of bytes of the pixel image that correspond to each row of the bitmap.
- n The **bitmap color space**—or the color space that determines how QuickDraw GX translates the bitmap's pixel values into colors. If the bitmap has any color space except indexed space, each pixel value in the pixel image represents a color specification in this color space. If the bitmap has an indexed color space, each pixel value is interpreted using the bitmap color set.
- n The **bitmap color set**—the optional array of color values associated with the bitmap. If the bitmap uses a color set (also called an *indexed color space*), each pixel value in the bitmap's pixel image represents an index into this color set.
- n The **bitmap color profile**—the color-matching information that you can specify for the device on which the bitmap was created.
- n The **bitmap position**—the position of the upper-left corner of the bitmap. The actual position of the bitmap when drawn may differ depending on the information in the bitmap's transform object.

QuickDraw GX provides the bitmap data type which you can use to create bitmap geometries. The bitmap data type has a field corresponding to every field of a bitmap geometry except the bitmap position. You must set and determine the bitmap position programmatically.

QuickDraw GX enforces a few restrictions on the values of these geometry fields. For example, the pixel size of the bitmap must be a power of 2 (from 1 to 32), and it must correspond to the pixel size implicit in the bitmap color space.

The bytes per row of the bitmap must be a multiple of 2. This requirement allows for faster bitmap manipulation.

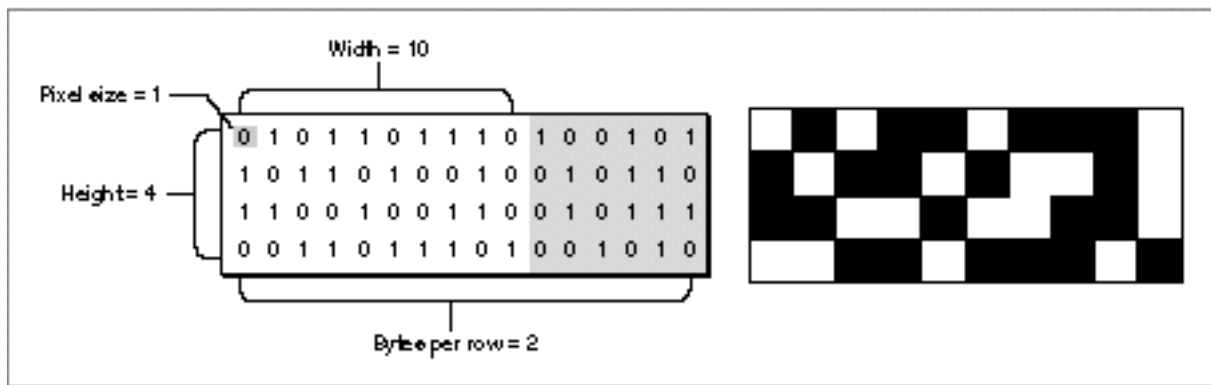
### Note

Although the Macintosh platform accepts any even number of bytes per row, you might want to use a multiple of 4 bytes per row in your bitmaps to promote cross-platform compatibility. u

## Bitmap Shapes

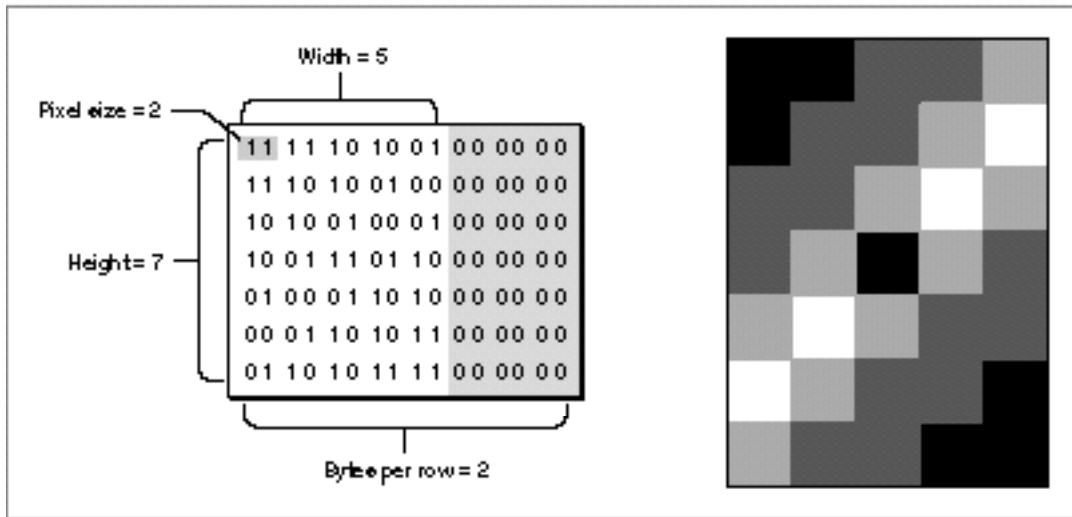
Sometimes you must pad a pixel image with extra bits to get an even number of bytes per row. For example, Figure 5-2 shows a small, black-and-white (1 bit per pixel) bitmap with a bitmap width of 10. The smallest number of bytes per row into which 10 bits fit is 2, so the number of bytes per row for this bitmap is 2. Although the six extra bits at the end of each row of the pixel image have values, they do not appear as pixels in the bitmap when it is drawn.

**Figure 5-2** A black-and-white bitmap geometry



All QuickDraw GX bitmaps are actually color bitmaps. A black-and-white bitmap is simply a color bitmap with a color set containing only two colors—black and white.

Figure 5-3 shows another bitmap. In this example, the pixel size is 2; each pixel is represented by 2 bits. Since no color space has an implicit pixel size of 2, this bitmap uses a color set instead of a color space. In this example, the color set contains four shades of gray (although it could contain any four colors); each pixel value in the pixel image is an index into this color set. Since the bitmap width is 5 pixels and the pixel size is 2 bits per pixel, at least 10 bits are required to represent each row of the pixel image. The smallest number of bytes per row that contains 10 bits is 2 bytes, so each row of the bitmap has 16 bits total. The last 6 bits are ignored by QuickDraw GX.

**Figure 5-3** A grayscale bitmap geometry

QuickDraw GX allows you to store a bitmap's pixel image in one of three locations:

- n You can allocate memory for the pixel image yourself. In this case, you provide QuickDraw GX with a pointer to the memory containing your pixel image. The section "Creating and Drawing Bitmaps" beginning on page 5-15 gives examples of allocating memory for a pixel image yourself and incorporating the pixel image into a bitmap shape.
- n You can request that QuickDraw GX allocate the memory for you. In this case, you must use QuickDraw GX functions to draw into the bitmap and to edit it. If you want to edit the pixel image directly, you can use other QuickDraw GX functions to lock the image in memory and to request a pointer to it. However, if QuickDraw GX is storing the pixel image on an accelerator card, your application might not be able to edit the pixel image directly. The section "Creating Bitmaps Offscreen" beginning on page 5-45 gives an example of requesting that QuickDraw GX allocate memory for your pixel image.
- n You can associate the bitmap with a **disk-based pixel image**—a pixel image stored in a disk file. In this case, you can use QuickDraw GX functions to read and draw the bitmap, but you cannot use QuickDraw GX functions to edit the bitmap. You must edit the bitmap directly using file-manipulation functions. The section "Creating Bitmaps With Disk-Based Pixel Images" beginning on page 5-44 shows how you can create a bitmap that uses a disk-based pixel image.

## Bitmap Styles and Inks

---

Although bitmap shapes have style objects and ink objects, they do not make full use of the properties of these objects. Of the many properties of the style object, only the style attributes property affects bitmap shapes. In fact, only the `gxSourceGridStyle` and `gxDeviceGridStyle` style attributes affect bitmap shapes.

QuickDraw GX ignores the other style attributes and the other style properties when drawing a bitmap shape. You can set the values of these properties and determine the values you have set them to, but they do not affect how the bitmap is drawn.

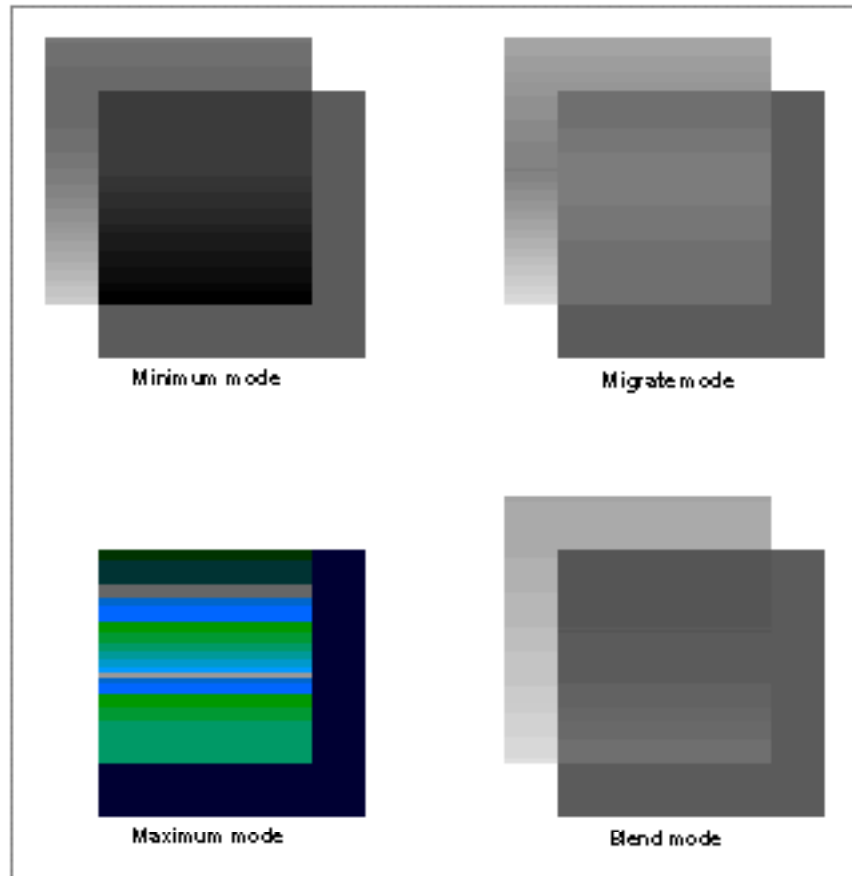
See Chapter 3, “Geometric Styles,” for a description of how the `gxSourceGridStyle` and `gxDeviceGridStyle` style attributes affect shapes, including bitmap shapes.

Of the ink object properties, bitmaps use the transfer mode property and ignore the color property. Since bitmap shapes have color information stored in their geometries, they do not need the color information stored in their ink objects. You can set the color of a bitmap shape’s ink object, but it does not affect how the bitmap is drawn.

The transfer mode property, on the other hand, does affect the drawing of the bitmap. QuickDraw GX applies the transfer mode as it draws each pixel of the bitmap, as shown in Figure 5-4. You can find a color version of this figure in Plate 1 at the front of this book.



**Figure 5-4** The effect of transfer modes on bitmap shapes

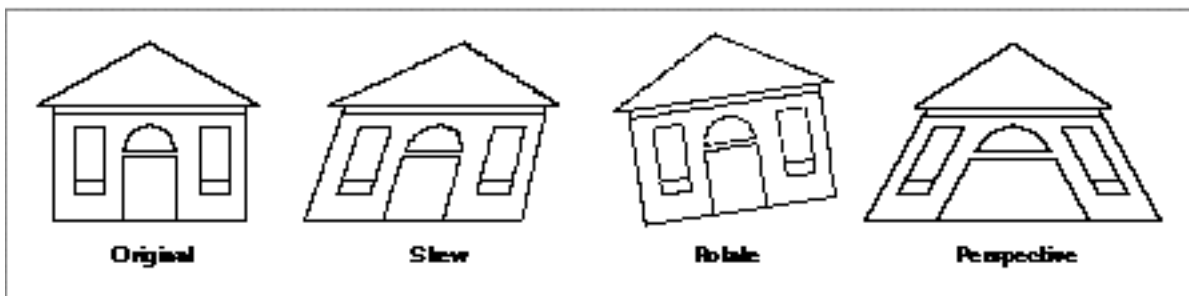


The section “Applying Transfer Modes to Bitmaps” beginning on page 5-32 shows you how to apply a transfer mode to a bitmap shape.

## Bitmap Transforms

Although bitmap shapes make limited use of their style and ink objects, they make full use of their transform objects. Using the transform object, you can clip bitmap shapes and apply mapping transformations to them. Some examples are shown in Figure 5-5.

**Figure 5-5** The effect of mappings on bitmap shapes

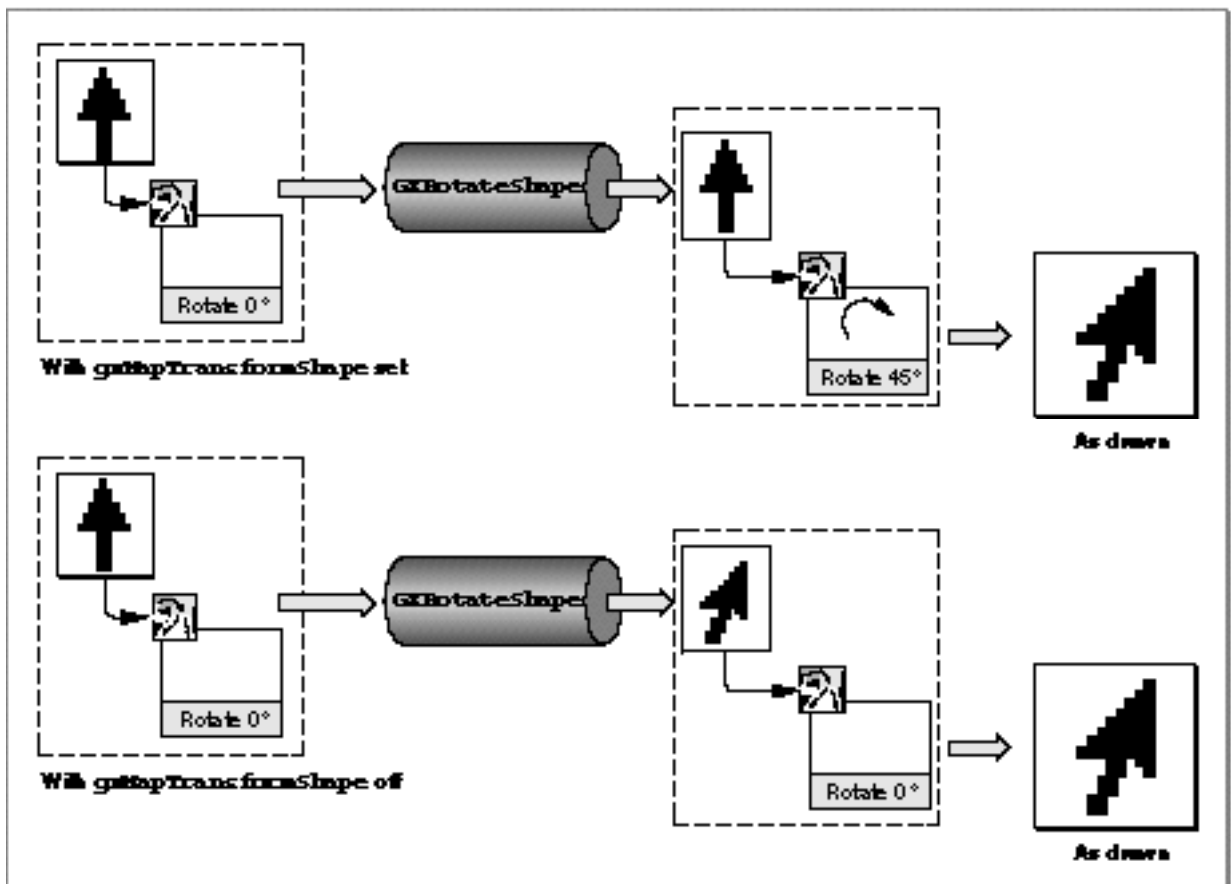


You can find examples of how to clip and map bitmap shapes in “Applying Transformations to Bitmaps,” which begins on page 5-38, and you can find more information about clipping and mapping in the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects* as well as the chapter “Mathematical Functions” in *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Bitmap shapes, like other types of shapes, use the `gxMapTransformShape` shape attribute to determine how mappings should be applied to the shape. If you set this shape attribute, applying a mapping to a bitmap shape changes the mapping matrix stored in the transform object of the bitmap shape. However, if you do not set this shape attribute, applying a mapping to a bitmap shape changes the geometry of the bitmap directly—that is, QuickDraw GX creates a completely new pixel image to represent the transformed bitmap.

Figure 5-6 compares the results of rotating a bitmap shape with and without the `gxMapTransformShape` shape attribute set.

**Figure 5-6** The effect of the `gxMapTransformShape` shape attribute on bitmap mappings



Each mapping that you apply to a bitmap shape that does not have its `gxMapTransformShape` shape attribute set results in quality degradation of the bitmap's pixel image. If you apply multiple mappings to a bitmap shape that does not have this shape attribute set, error can arise rapidly. The section "Applying Transformations to Bitmaps," which begins on page 5-38, gives an example of this phenomenon.

You can find information about the `GXRotateShape` function in the chapter "Transform Objects" in *Inside Macintosh: QuickDraw GX Objects*.

## Bitmaps and View Devices

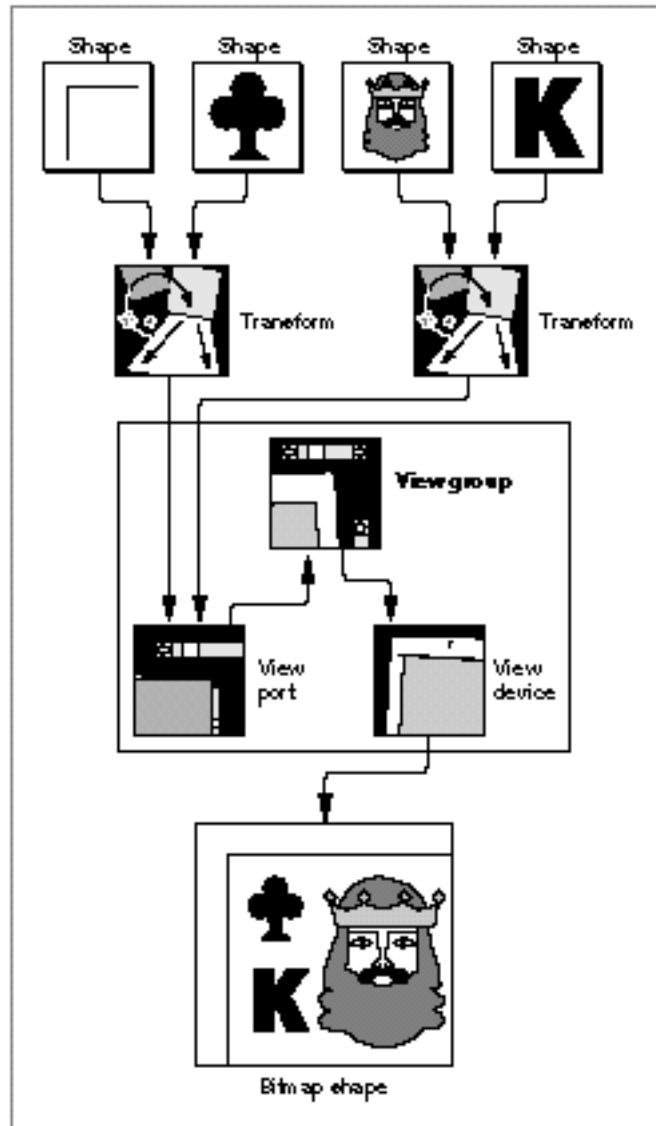
---

When you draw a shape, QuickDraw GX applies the information in the shape's style, ink, and transform objects to the shape's geometry and then renders the shape to the display devices that correspond to the view information contained in the shape's transform object.

The transform object of a shape contains a list of view ports to which QuickDraw GX should draw the shape. Each view port exists in the coordinate space of a specific view group, and each view group contains view devices that share the same coordinate space. QuickDraw GX determines where the shape appears in the coordinate space of each view group. If the area of the shape when drawn overlaps the area covered by any view device in that view group, QuickDraw GX renders the shape into the bitmap attached to that view device.

Figure 5-7 depicts how this drawing mechanism works with four shapes: a polygon shape, a path shape, a bitmap shape, and a text shape. The two path shapes share one transform object and the bitmap shape and the text shape share a second transform object.

Both of the transform objects contain one view port in their view port list. That view port exists in a view group that also contains a view device. The view device has a bitmap shape associated with it to hold the renderings of shapes drawn to it.

**Figure 5-7** Bitmaps and view devices

## Bitmap Shapes

Whenever you draw a QuickDraw GX shape, you are using this view architecture to render the shape to a display device. You can also use this view architecture to draw shapes into an **offscreen bitmap**—a bitmap that is not associated with a physical display device.

The section “Creating Bitmaps Offscreen,” which begins on page 5-45, shows how you can create an offscreen bitmap, draw shapes into it, and then draw it to the screen.

You can find more information about the QuickDraw GX view architecture in the chapter “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## Using Bitmap Shapes

---

This section shows you how to create, edit, and draw bitmap shapes. In particular, this section shows you how to

- n create and draw black-and-white bitmaps
- n create and draw color bitmaps
- n dither and halftone bitmaps
- n apply transfer modes to bitmaps
- n convert other types of shapes to bitmap shapes
- n apply transformations to bitmaps
- n create offscreen bitmaps
- n edit sections of bitmaps

Bitmap shape geometries use a `gxPoint` structure to indicate the initial position of the bitmap. Since a `gxPoint` structure contains two fixed-point values (type `Fixed`), the sample functions in this section must convert integer constants to fixed-point constants when specifying bitmap positions. QuickDraw GX provides the `GXIntToFixed` macro to perform this conversion:

```
#define GXIntToFixed(a) ((Fixed)(a) << 16)
```

QuickDraw GX also provides the `ff` macro as a convenient alias:

```
#define ff(a) GXIntToFixed(a)
```

## Creating and Drawing Bitmaps

---

QuickDraw GX provides a number of methods to create and draw bitmaps. For example, you can

- n define a bitmap geometry and draw it without creating a bitmap shape
- n define a bitmap geometry, encapsulate it in a bitmap shape, and draw the bitmap shape
- n create another type of shape, convert it to a bitmap shape, perform any desired bitmap editing, and draw the bitmap shape
- n create an offscreen bitmap, draw shapes to it, and then copy the offscreen bitmap to the screen
- n unflatten a bitmap shape that was created earlier and stored to disk or that was created by another application
- n convert a QuickDraw bitmap to a QuickDraw GX bitmap shape

The next section, “Creating Black-and-White Bitmaps,” and “Creating Color Bitmaps,” which begins on page 5-21, show you how to create bitmaps by specifying the bitmap geometry yourself.

The section “Converting Other Types of Shapes to Bitmaps,” which begins on page 5-34, shows you how you can create a bitmap shape containing a bitmap representation of other types of QuickDraw GX shapes.

The section “Creating Bitmaps Offscreen,” which begins on page 5-45, shows you how you can draw other shapes into the pixel image of a bitmap shape. You can use this method to create a bitmap representation of multiple QuickDraw GX shapes.

For information about flattening and unflattening bitmap shapes, see the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## Creating Black-and-White Bitmaps

---

You create a black-and-white bitmap by creating a bitmap shape with a pixel size of 1. To do this, you can define a pixel image, fill the fields of a bitmap geometry structure, and create a bitmap shape using the `GXNewBitmap` function.

Listing 5-1 shows a complete sample function that defines a black-and-white bitmap geometry, creates a bitmap shape, draws the shape, and disposes of it.

---

**Listing 5-1**      Creating a black-and-white bitmap

```
void CreateBlackAndWhiteBitmap(void)
{
    gxShape  aBitmapShape;

    gxBitmap aBitmapGeometry;
    gxPoint  initialPosition = {ff(20), ff(40)};
```

## Bitmap Shapes

```

const char envelopeImage[] = {0x7F, 0xFF, 0xFF, 0xFE,
                               0xC0, 0x00, 0x00, 0x03,
                               0xB0, 0x00, 0x00, 0x0D,
                               0x8C, 0x00, 0x00, 0x31,
                               0x83, 0x00, 0x00, 0xC1,
                               0x80, 0xC0, 0x03, 0x01,
                               0x80, 0x30, 0x0C, 0x01,
                               0x80, 0x0C, 0x30, 0x01,
                               0x80, 0x33, 0xCC, 0x01,
                               0x80, 0xC0, 0x03, 0x01,
                               0x83, 0x00, 0x00, 0xC1,
                               0x8C, 0x00, 0x00, 0x31,
                               0xB0, 0x00, 0x00, 0x0D,
                               0x7F, 0xFF, 0xFF, 0xFE};

aBitmapGeometry.image = (char *) aSmallBitmapImage;

aBitmapGeometry.width = 32;      /* width in pixels */
aBitmapGeometry.height = 14;     /* height in pixels */
aBitmapGeometry.rowBytes = 4;    /* bytes per row */
aBitmapGeometry.pixelSize = 1;   /* bits per pixel */

/* QuickDraw GX creates a black-and-white color set for you */
aBitmapGeometry.space = gxNoSpace;
aBitmapGeometry.set = nil;
aBitmapGeometry.profile = nil;

aBitmapShape = GXNewBitmap(&aBitmapGeometry, &initialPosition);

GXDrawShape(aBitmapShape);
GXDisposeShape(aBitmapShape);
}

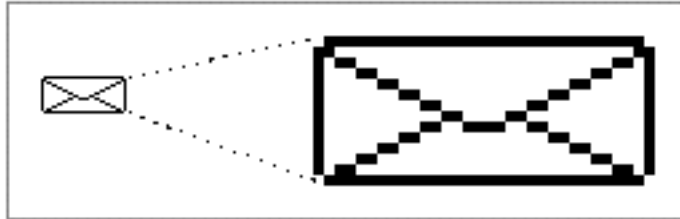
```



## Bitmap Shapes

The result of this function is shown in Figure 5-8.

**Figure 5-8** A black-and-white bitmap—32 bits wide



The sample function from Listing 5-1 first defines a variable to hold the reference to the bitmap shape:

```
gxShape aBitmapShape;
```

Then the sample function defines two local variables to specify the bitmap geometry:

```
gxBitmap aBitmapGeometry;

gxPoint initialPosition = {ff(20), ff(40)};
```

The `initialPosition` variable, which is type `gxPoint`, contains the initial bitmap position, and the `aBitmapGeometry` variable, which is type `gxBitmap`, contains the rest of the information about the bitmap.

The sample function then defines the bitmap's pixel image:

```
const char envelopeImage[] = {0x7F, 0xFF, 0xFF, 0xFE,
                              0xC0, 0x00, 0x00, 0x03,
                              0xB0, 0x00, 0x00, 0x0D,
                              0x8C, 0x00, 0x00, 0x31,
                              0x83, 0x00, 0x00, 0xC1,
                              0x80, 0xC0, 0x03, 0x01,
                              0x80, 0x30, 0x0C, 0x01,
                              0x80, 0x0C, 0x30, 0x01,
                              0x80, 0x33, 0xCC, 0x01,
                              0x80, 0xC0, 0x03, 0x01,
                              0x83, 0x00, 0x00, 0xC1,
                              0x8C, 0x00, 0x00, 0x31,
                              0xB0, 0x00, 0x00, 0x0D,
                              0x7F, 0xFF, 0xFF, 0xFE};
```

## Bitmap Shapes

The `envelopeImage` variable, which is defined as an array of bytes, contains a pixel image depicting a small envelope, as shown in Figure 5-8.

To create a bitmap shape encapsulating this envelope image, the sample function fills in the eight fields of the `aBitmapGeometry` variable. First, it sets the `image` field by casting the `envelopeImage` variable to the correct type:

```
aBitmapGeometry.image = (char *) envelopeImage;
```

Then the sample function fills in the bitmap dimensions. The bitmap is 32 pixels wide by 14 pixels high, and there are 4 bytes of information in each row of the pixel image:

```
aBitmapGeometry.width = 32;           /* width in pixels */
aBitmapGeometry.height = 14;          /* height in pixels */
aBitmapGeometry.rowBytes = 4;         /* bytes per row */
```

The sample function specifies the pixel size next. Since this bitmap is black-and-white, only one bit is needed to represent each pixel of the bitmap:

```
aBitmapGeometry.pixelSize = 1;        /* bits per pixel */
```

Finally, the sample function specifies color information. Since QuickDraw GX does not provide a black-and-white color space, this bitmap needs a black-and-white color set in which pixel values of 0 represent white pixels and pixel values of 1 represent black pixels. Setting the `pixelSize` field to 1 and the `space` field to `gxNoSpace` indicates that QuickDraw GX should create this black-and-white color set for you.

```
aBitmapGeometry.space = gxNoSpace;
aBitmapGeometry.set = nil;
aBitmapGeometry.profile = nil;
```

Setting the `space` field to the value `gxNoSpace` always indicates that QuickDraw GX should choose a color space for you. If the pixel size were large—for example, 16 or 32—QuickDraw GX would choose an RGB color space. However, since the pixel size is 1, no appropriate color space exists, so QuickDraw GX creates a grayscale color set. The pixel size determines the size of the color set created. In this case, a pixel size of 1 dictates that the color set have two entries—an white entry for a pixel value of 0 and a black entry for a pixel value of 1.

After you define a bitmap geometry, you could use the `GXDrawBitmap` function to cause QuickDraw GX to

- n create a temporary bitmap shape (using the style, ink, and transform objects of the default bitmap shape)
  - n draw the bitmap
  - n dispose of the temporary bitmap shape
- with this line of code:

```
GXDrawBitmap(&aBitmapGeometry, &initialPosition);
```

## Bitmap Shapes

You should use the `GXDrawBitmap` function, however, only when you know in advance that you want to draw a bitmap only one time.

If you want to draw a bitmap more than once, you should encapsulate the bitmap geometry in a bitmap shape and then draw the bitmap shape. The sample function in Listing 5-1 uses this method:

```
aBitmapShape = GXNewBitmap(&aBitmapGeometry, &initialPosition);
GXDrawShape(aBitmapShape);
```

As with any type of QuickDraw GX shape, if you create a bitmap shape, you are responsible for disposing of it when you no longer need it. Listing 5-1 does this by calling

```
GXDisposeShape(aBitmapShape);
```

Notice that the envelope bitmap requires 4 bytes—an even number—to represent each row of the pixel image. However, to draw a similar envelope bitmap that includes two more rows of bits, as shown in Figure 5-10, the required number of bytes might seem to be 5 since 5 bytes contain 40 bits, more than enough needed to store the 34 bits per row in this image.

However, if you set the `rowBytes` field to 5:

```
aBitmapGeometry.rowBytes = 5;
```

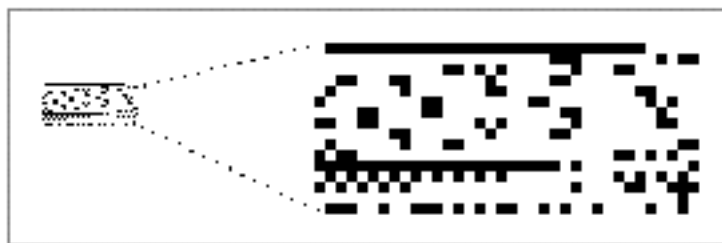
both the `GXDrawBitmap` function and the `GXNewBitmap` function post the error `bitmap_rowBytes_not_aligned`, because the value of the `rowBytes` field must be an even number.

Therefore, the value of the `rowBytes` field must be at least 6 for the bitmap of the envelope with a shadow. However, simply setting the `rowBytes` field to the value 6 with the assignment

```
aBitmapGeometry.rowBytes = 6;
```

results in the bitmap shown in Figure 5-9.

**Figure 5-9** An example of unaligned bytes per row



## Bitmap Shapes

Clearly, the value of the bitmap's `rowBytes` field is not aligned with the data in the bitmap's pixel image. If you set the value of the `rowBytes` field to 6, you must be sure to pad the pixel image so that each row actually contains 6 bytes of information. Listing 5-2 shows a new definition of the pixel image. In this definition, each row contains one extra byte so that the total number of bytes per row is even.

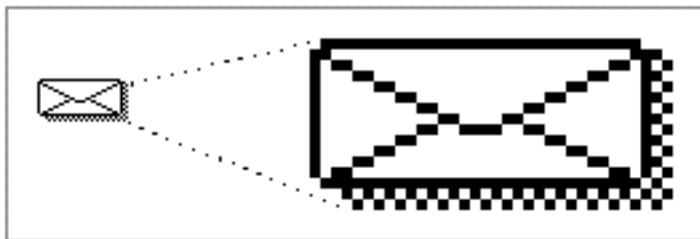
In this example, the extra bytes are initialized to the value 0x00. However, since these bytes are just padding, you can specify any values for them. As indicated by the bitmap width, QuickDraw GX ignores these extra bytes when drawing, hit-testing, or otherwise manipulating the bitmap.

**Listing 5-2** A bit image with an even number of bytes per row

```
static char envelopeImage[] = {0x7F, 0xFF, 0xFF, 0xFE, 0x00, 0x00,
                               0xC0, 0x00, 0x00, 0x03, 0x50, 0x00,
                               0xB0, 0x00, 0x00, 0x0D, 0xA0, 0x00,
                               0x8C, 0x00, 0x00, 0x31, 0x50, 0x00,
                               0x83, 0x00, 0x00, 0xC1, 0xA0, 0x00,
                               0x80, 0xC0, 0x03, 0x01, 0x50, 0x00,
                               0x80, 0x30, 0x0C, 0x01, 0xA0, 0x00,
                               0x80, 0x0C, 0x30, 0x01, 0x50, 0x00,
                               0x80, 0x33, 0xCC, 0x01, 0xA0, 0x00,
                               0x80, 0xC0, 0x03, 0x01, 0x50, 0x00,
                               0x83, 0x00, 0x00, 0xC1, 0xA0, 0x00,
                               0x8C, 0x00, 0x00, 0x31, 0x50, 0x00,
                               0xB0, 0x00, 0x00, 0x0D, 0xA0, 0x00,
                               0x7F, 0xFF, 0xFF, 0xFE, 0x50, 0x00,
                               0x15, 0x55, 0x55, 0x55, 0xA0, 0x00,
                               0x0A, 0xAA, 0xAA, 0xAA, 0x80, 0x00};
```

With this new, padded definition of the pixel image, you can set `rowBytes` field to 6 so that the resulting bitmap appears as shown in Figure 5-10.

**Figure 5-10** An envelope with a shadow



## Bitmap Shapes

For a discussion of pixel images and bitmap geometries, see “Bitmap Geometries” beginning on page 5-5.

For more information about the `GXNewBitmap` function, see its description on page 5-66. For more information about the `GXDrawBitmap` function, see its description on page 5-77.

The next section shows you how you can create a bitmap with color information.

### Creating Color Bitmaps

---

All QuickDraw GX bitmaps are actually color bitmaps. A black-and-white bitmap is simply a color bitmap with a color set containing only two colors—black and white.

The sample function in Listing 5-1 on page 5-15 creates a black-and-white bitmap geometry by

- n specifying the pixel size to be 1
- n specifying the color space to be the `gxNoSpace` color space

The sample function encapsulates the geometry into a bitmap shape with this call to the `GXNewBitmap` function:

```
aBitmapShape = GXNewBitmap(&aBitmapGeometry, &initialPosition);
```

Because the `space` field of the bitmap geometry specifies the `gxNoSpace` color space, the `GXNewBitmap` function chooses a color space for you, based on the pixel size specified in the `pixelSize` field. QuickDraw GX does not provide any color spaces appropriate for a pixel size of 1, so the `GXNewBitmap` function creates a grayscale color set with two entries—white and black.

If you specify the `gxNoSpace` color space with a pixel size of 2, the `GXNewBitmap` function creates a grayscale color set with four entries—white, light gray, dark gray, and black. After you change the pixel size to 2, you must reflect that change in the pixel image, the bitmap width, and the number of bytes per row.

Typically, if you wanted to make a 1 bit-per-pixel bitmap into a 2 bit-per-pixel bitmap, you would do the following

- n Maintain the bitmap width, as it represents the number of pixel values—not the number of bits—per row of the bitmap
- n Double the number of bytes per row to accommodate the extra bits
- n Double the size of the pixel image, replacing 1-bit pixel values with 2-bit pixel values

This method allows you to maintain the size of the bitmap while allowing you to specify more possible values (colors) for each pixel.

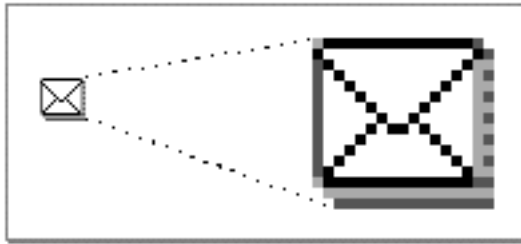
## Bitmap Shapes

However, an easier (if somewhat less useful) way to make a 1 bit-per-pixel bitmap into a 2 bit-per-pixel bitmap is as follows:

- n Divide the bitmap width in half
- n Maintain the same number of bytes per row
- n Maintain the same pixel image

Let's see what happens when you apply this simpler method for doubling the pixel size of a bitmap. Doubling the pixel size and halving the bitmap width of the envelope bitmap shown in Figure 5-10 on page 5-20 indicates that QuickDraw GX should interpret every pair of bits in the pixel image as a single pixel. Since each pixel can have one of four possible values (00, 01, 10, 11), the resulting bitmap contains four shades of gray, as shown in Figure 5-11.

**Figure 5-11** A bitmap with a grayscale color set (four shades)



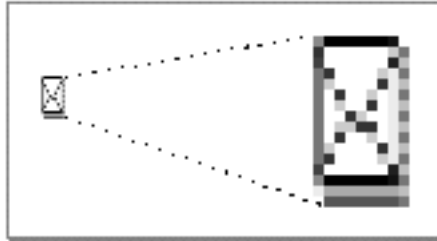
You can double the pixel size and halve the bitmap width again, with the following assignments:

```
aBitmapGeometry.width = 9;
aBitmapGeometry.height = 16;
aBitmapGeometry.rowBytes = 6;
aBitmapGeometry.pixelSize = 4;
```

QuickDraw GX interprets each set of 4 bits in the pixel image as representing a single pixel of the bitmap, which means each pixel can now be represented by 16 different values (0000, 0001, 0010, and so on). Since QuickDraw GX has no predefined color space that uses a pixel size of 4, it creates for this bitmap a color set with sixteen shades of gray.

Figure 5-12 shows the resulting bitmap.

**Figure 5-12** A bitmap with a grayscale color set (sixteen shades)



As the previous examples have shown, setting the `space` field of a bitmap geometry to the `gxNoSpace` constant indicates that you want QuickDraw GX to choose a color space for you. In these examples, which had 1, 2, or 4 bits per pixel, QuickDraw GX chose the `gxIndexedSpace` color space and created a grayscale color set with the appropriate number of color entries.

You are not limited to these grayscale color sets, however. You can create your own color set, by choosing your own set of colors for the color entries. Listing 5-3 shows how to define a simple color set with eight colors—black and white, the three primary RGB colors, and the three secondary RGB colors.

**Listing 5-3** Defining a color set

```
gxColorSet aColorSet;

gxSetColor newColorList[] = {
    {0xFFFF, 0xFFFF, 0xFFFF, 0},    /* white */
    {0xFFFF, 0,      0,      0},    /* red */
    {0,      0xFFFF, 0,      0},    /* green */
    {0,      0,      0xFFFF, 0},    /* blue */
    {0,      0xFFFF, 0xFFFF, 0},    /* cyan */
    {0xFFFF, 0,      0xFFFF, 0},    /* magenta */
    {0xFFFF, 0xFFFF, 0,      0},    /* yellow */
    {0,      0,      0,      0},    /* black */
};
```

## Bitmap Shapes

The colors in this color set are specified in the RGB color space, and each color contains four components—the red component, the green component, the blue component, and a fourth component, which QuickDraw GX ignores for the RGB color space.

QuickDraw GX allows you to specify colors in other color spaces and with different numbers of components. For complete color-specifying information, see the chapter “Colors and Color-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

Once you’ve defined the color list for a color set, you create the actual color set object by using the `GXNewColorSet` function, which requires you to specify the color space in which you’ve specified the colors, the total number of colors, and the list of colors:

```
aColorSet = GXNewColorSet(gxRGBSpace, 8, newColorList);
```

**Note**

Remember, you are responsible for disposing of QuickDraw GX objects when you no longer need them, so you are responsible for disposing of this new color set. <sup>u</sup>

To use the new color set in your bitmap, you need to set the `space` and `set` fields of the bitmap geometry:

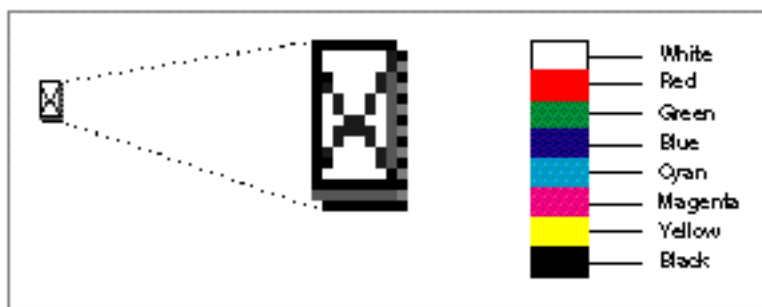
```
aBitmapGeometry.space = gxIndexedSpace;  
aBitmapGeometry.set = aColorSet;
```

Setting the `space` field to `gxIndexedSpace` indicates that you are supplying the color set, rather than having QuickDraw GX create one for you.

Figure 5-13 shows the result of applying this new color set to the 4-bits-per-pixel version of the envelope bitmap. Notice that pixel values in the pixel image greater than 7 are out of the range of the color set, so QuickDraw GX maps those pixel values to the color black.

For a color version of Figure 5-13, see Plate 3 at the front of this book.

**Figure 5-13** A bitmap with an eight-color color set





## Bitmap Shapes

Each of the previous examples in this chapter creates a bitmap that uses a color set. QuickDraw GX interprets each pixel of these bitmaps as an index into a set of colors. For example, in the black-and-white bitmap that results from Listing 5-1 on page 5-15, each pixel value (single bit) of the pixel image is an index into a color set with two colors—the index of the color white is 0 and the index of the color black is 1. In the 2 bits-per-pixel example on page 5-6, each pixel value (pair of bits) in the pixel image is an index into a color set with four colors—the index of white is 0 (bits 00), the index of light gray is 1 (bits 01), the index of dark gray is 2 (bits 10), and the index of black is 3 (bits 11).

QuickDraw GX also allows you to create bitmaps that use color spaces other than indexed color spaces (that is, other than color sets). In these bitmaps, each pixel value is an actual color value instead of an index into a list of colors. The chapter “Colors and Color-Related Objects” in *Inside Macintosh: QuickDraw GX Objects* explains color spaces, color values, color sets, and color indexes.

One example of a bitmap for which you might want to use a color space instead of a color set is a color ramp. A **color ramp** is a shape that blends from one color into another. Since bitmaps are the only type of QuickDraw GX shape (except picture shapes) that allows multiple colors in one shape, you must implement color ramps as bitmap shapes.

The sample function in Listing 5-4 on page 5-26 shows how to create a simple color ramp. This function declares a bitmap shape reference and a bitmap geometry structure using the declarations

```
gxShape  aBitmapShape;
gxBitmap aBitmapGeometry;
```

It then fills in the fields of the bitmap geometry structure. First, it fills in the dimensions:

```
aBitmapGeometry.width = 1;
aBitmapGeometry.height = 256;
aBitmapGeometry.rowBytes = 1;
aBitmapGeometry.pixelSize = 32;
```

Notice that the sample function defines the bitmap width to be 1. Later, the sample function uses the `GXSetShapeBounds` function later to widen the bitmap.

Next, the sample function sets the `image` field to `nil` to indicate that QuickDraw GX should allocate memory for the pixel image of the bitmap. The value of the `rowBytes` field is ignored because QuickDraw GX sets this field when allocating the pixel image.

The sample function then sets the color-related fields of the bitmap geometry structure:

```
aBitmapGeometry.space = gxRGB32Space;
aBitmapGeometry.set = nil;
aBitmapGeometry.profile = nil;
```

Notice that the pixel size implied by the color space (which is the `gxRGB32Space` color space) is the same as the pixel size indicated in the `pixelSize` field of the bitmap geometry structure (which is 32).

## Bitmap Shapes

Next, the sample function creates the bitmap shape:

```
aBitmapShape = GXNewBitmap(&aBitmapGeometry, &initialPosition);
```

The sample function sets the color values of each pixel in the bitmap shape. To do this, it creates a color structure with the declaration

```
gxColor current;
```

Then it fills in the values of the fields of the color structure:

```
current.space = gxRGBSpace;
current.profile = nil;
current.element.rgb.red = 0xFFFF;
current.element.rgb.green = 0;
current.element.rgb.blue = 0;
current.element.rgb.alpha = 0;
```

For a complete discussion of these fields, see the chapter “Colors and Color-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

The sample function then uses the `GXSetShapePixel` function to set each pixel value in the pixel image of the bitmap shape. Each time the sample function sets the value of a pixel, it changes the color value of the `current` variable slightly, decreasing the amount of green and increasing the amount of red:

```
for (count = 0; count < 256; count++) {
    current.element.component[0] -= 0x0101; /* decrease red */
    current.element.component[1] += 0x0101; /* increase green */

    GXSetShapePixel(aBitmapShape, 0, count, &current, 0);
}
```

Finally, the sample function resizes the bitmap, widening it to be a square, and draws the resulting bitmap color ramp. The complete sample function definition is shown in Listing 5-4.

---

**Listing 5-4**      Creating a color ramp

```
void CreateColorRamp(void)
{
    gxShape aBitmapShape;
    gxBitmap aBitmapGeometry;
    const gxPoint initialLocation = {ff(50), ff(50)};
    const gxRectangle theBounds = {ff(50), ff(50),
                                   ff(150), ff(150)};
```

## Bitmap Shapes

```

gxColor current;
int count;

/* create a one-pixel-wide bitmap */
aBitmapGeometry.width = 1;
aBitmapGeometry.height = 256;
aBitmapGeometry.rowBytes = 1;
aBitmapGeometry.pixelSize = 32;

aBitmapGeometry.image = nil; /* have QuickDraw GX allocate */

aBitmapGeometry.space = gxRGB32Space;
aBitmapGeometry.set = nil;
aBitmapGeometry.profile = nil;

aBitmapShape = GXNewBitmap(&aBitmapGeometry, &initialLocation);

/* create a red color */
current.space = gxRGBSpace;
current.profile = nil;
current.element.component[0] = 0xFFFF; /* red */
current.element.component[1] = 0;      /* green */
current.element.component[2] = 0;      /* blue */
current.element.component[3] = 0;      /* alpha */

/* fill in the colors of the bitmap pixel by pixel */
for (count = 0; count < 256; count++) {
    current.element.rgb.red -= 0x0101; /* decrease red */
    current.element.rgb.green += 0x0101; /* increase green */

    GXSetShapePixel(aBitmapShape, 0, count, &current, 0);
}

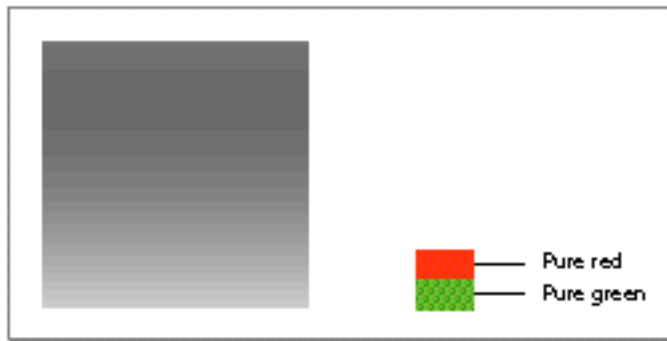
/* resize the bitmap to give it more width */
GXSetShapeBounds(aBitmapShape, &theBounds);

GXDrawShape(aBitmapShape);
GXDisposeShape(aBitmapShape);
}

```

The resulting color ramp is shown in Figure 5-14. For a color version of this figure, see Plate 4 at the front of this book.

**Figure 5-14** A color ramp from red to green



QuickDraw GX provides the ramp library to assist you in creating color ramps. The `NewRamp` library function requires you to provide a start color, an end color, an integer indicating the number of different colors to calculate in between the start color and end color, and a bounding rectangle for the final color ramp. Listing 5-5 shows how to use this function to create the same color ramp shown in Figure 5-14.

**Listing 5-5** Creating a color ramp using the ramp library

```
void CreateColorRamp(void)
{
    gxShape aBitmapShape;
    gxColor start, end;
    const gxRectangle theBounds = {ff(50), ff(50),
                                    ff(150), ff(150)};

    start.space = gxRGBSpace;
    start.profile = nil;
    start.element.rgb.red = 0xFFFF;
    start.element.rgb.green = 0;
    start.element.rgb.blue = 0;
    start.element.rgb.alpha = 0;

    end.space = gxRGBSpace;
    end.profile = nil;
    end.element.rgb.red = 0;
    end.element.rgb.green = 0xFFFF;
}
```

## Bitmap Shapes

```

    end.element.rgb.blue = 0;
    end.element.rgb.alpha = 0;

    aBitmapShape = NewRamp(&start, &end, 256, &theBounds);
    GXDrawShape(aBitmapShape);
    GXDisposeShape(aBitmapShape);
}

```

As a further convenience, QuickDraw GX provides the color library, which allows you to use predefined constants to specify frequently used colors. You provide the `SetCommonColor` library function with a pointer to a color structure, and a predefined constant specifying the color you want.

This function then initializes the color structure with the appropriate values to represent the color you specify.

For example, the following call sets the fields of the `start` color structure with the values that represent the color red:

```
SetCommonColor(&start, red);
```

Listing 5-6 shows you how to create the color ramp in Figure 5-14 by using functions from both the ramp and color libraries.

---

**Listing 5-6**      Creating a color ramp using both the ramp and color libraries

```

void CreateColorRamp(void)
{
    gxShape  aBitmapShape;
    gxColor  start, end;
    const gxRectangle theBounds = {ff(50), ff(50),
                                   ff(150), ff(150)};

    SetCommonColor(&start, red);
    SetCommonColor(&end, green);

    aBitmapShape = NewRamp(&start, &end, 0, &theBounds);
    GXDrawShape(aBitmapShape);
    GXDisposeShape(aBitmapShape);
}

```

For a discussion of pixel images and bitmap geometries, see “Bitmap Geometries” beginning on page 5-5.

You can find more information about colors, color structures, color values, color sets, and color spaces in the chapter “Colors and Color-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## Dithering and Halftoning Bitmaps

---

The color ramp created in the previous section uses the `gxRGB32Space` color space, but not all display devices can display 32 bits of color. To optimize the appearance of color on displays with limited numbers of colors, QuickDraw GX allows you to **dither** shapes—that is, approximate colors that a display device cannot draw, with patterns of similar colors that the display device can draw.

The chapter “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects* describes dithering in detail.

This section shows how you can use dithering to draw the color ramp shown in Figure 5-14 on page 5-28.

Since dithering is a function of view port objects, you must first determine the view port to which the color ramp is drawn. Since this color ramp is only being drawn to one view port, you can declare an array to hold a single view port reference:

```
gxViewPort aViewPortList[1];
```

Then you can use the `GXGetShapeGlobalViewPorts` function to copy the view port list from the transform object of the color ramp bitmap shape into the view port array:

```
GXGetShapeGlobalViewPorts(aColorRampBitmapShape, aViewPortList);
```

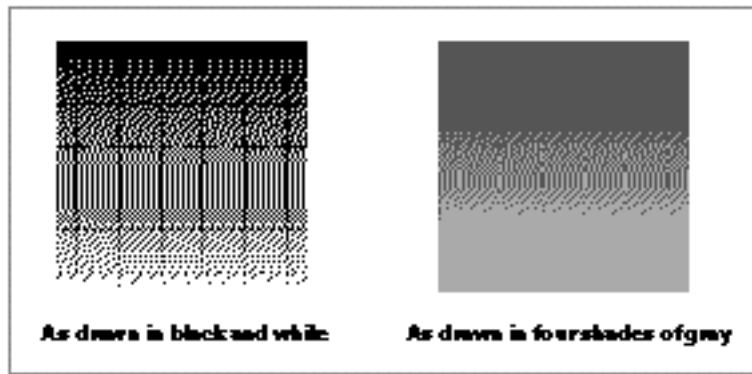
If the color ramp were being drawn to multiple view ports, you would call this function once specifying `nil` for the view port array to determine the number of view ports, then allocate space to hold the view port references, and then call the function a second time to determine the actual view port references.

In the color ramp example, you can use the `GXSetViewPortDither` function to indicate that shapes drawn to this view port should be dithered. This function takes two parameters: a reference to the view port and a dither level, which is described in detail in the chapter “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*. If a view port has a dither level of 2 or greater, QuickDraw GX dithers bitmaps drawn to that view port:

```
GXSetViewPortDither(aViewPortList[0], 4); /* Dither bitmaps */
```

Figure 5-15 shows how QuickDraw GX draws the dithered color ramp to display devices at two different pixel depths.

**Figure 5-15** Dithered bitmaps



**Halftoning**, which is also described in the chapter “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*, is similar to dithering. To specify halftoning for a view port, you need to create a `gxHalftone` structure. This structure specifies information about how QuickDraw GX should halftone shapes drawn to the view port. Listing 5-7 shows how to create a sample `gxHalftone` structure and set the halftone characteristics for the view port of the color ramp bitmap.

**Listing 5-7** Halftoning a bitmap

```
gxHalftone aHalftone;

SetCommonColor(&halftoneDots, gxBlack);
SetCommonColor(&halftoneBackground, gxWhite);

aHalftone.angle = ff(45);
aHalftone.frequency = ff(5);
aHalftone.method = gxRoundDot;
aHalftone.tinting = gxComponent1Tint;
aHalftone.dotColor = halftoneDots;
aHalftone.backgroundColor = halftoneBackground;
aHalftone.tintSpace = gxRGBSpace;

GXGetShapeGlobalViewPorts(aBitmapShape, aViewPort);
GXSetViewPortHalftone(aViewPort[0], &aHalftone);
```

## Bitmap Shapes

Figure 5-16 shows three possible results of halftoning the color ramp bitmap. The first example is the result of Listing 5-7—round dots and a dot frequency of 5. The other two examples show the result of halftoning the color ramp bitmap with other dot frequencies and dot shapes.

**Figure 5-16** Halftoned bitmaps



### Applying Transfer Modes to Bitmaps

When drawing a bitmap, QuickDraw GX uses the color information stored in the geometry of the bitmap shape; it ignores the color information stored in the ink object associated with the bitmap shape.

However, QuickDraw GX does consider the transfer mode information specified in a bitmap shape's ink object. QuickDraw GX uses the transfer mode when drawing each pixel of a bitmap.



## Bitmap Shapes

As an example, the sample function in Listing 5-8 creates a rectangle shape containing a purple rectangle and a bitmap shape containing a color ramp from red to green (as defined in Listing 5-6 on page 5-29).

---

**Listing 5-8** Applying a transfer mode to a bitmap

```
void ApplyTransferModeToBitmap(void)
{
    gxShape aRectangleShape, aBitmapShape;
    const gxRectangle theRectangleBounds = {ff(100), ff(100),
                                            ff(200), ff(200)};

    const gxRectangle theBitmapBounds = {ff(50), ff(50),
                                         ff(150), ff(150)};

    gxColor start, end;

    aRectangleShape = GXNewRectangle(&theRectangleBounds);
    SetShapeCommonColor(aRectangleShape, purple);

    SetCommonColor(&start, red);
    SetCommonColor(&end, green);
    aBitmapShape = NewRamp(&start, &end, 0, &theBitmapBounds);
    SetShapeCommonTransfer(aBitmapShape, gxBlendMode);

    GXDrawShape(aRectangleShape);
    GXDrawShape(aBitmapShape);

    GXDisposeShape(aRectangleShape);
    GXDisposeShape(aBitmapShape);
}
```

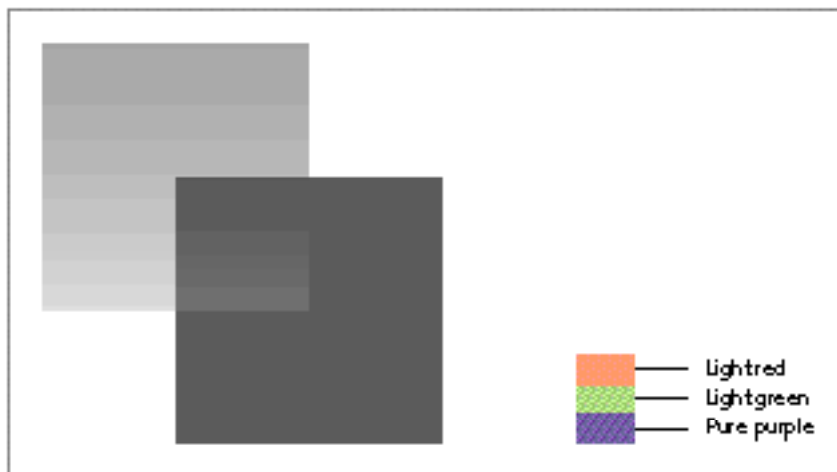
## Bitmap Shapes

The sample function then uses the transfer mode library function `SetShapeCommonTransfer` to set the transfer mode of the bitmap shape to `gxBlendMode`.

Finally, the sample function draws the purple rectangle and the bitmap. Since the ink object associated with the bitmap specifies the `gxBlendMode` transfer mode, QuickDraw GX applies this transfer mode when drawing each pixel of the bitmap. Pixels that fall over the white background are blended with white, and pixels that fall over the purple rectangle are blended with purple.

Figure 5-17 shows the result of this sample function. For a color version of this figure, see Plate 2 at the front of this book.

**Figure 5-17** A blended color ramp



You can find more information about transfer modes in the chapter “Ink Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## Converting Other Types of Shapes to Bitmaps

The examples in the previous sections show you how to create a bitmap shape by specifying the value of every pixel in the pixel image yourself. You can also create bitmaps using one of a number of simpler methods. For example, you can convert any QuickDraw GX shape to a bitmap shape. The pixel image of the resulting bitmap shape contains a bitmap representation of the original shape. (In a similar way, when you draw a shape to a display device, the display device displays a bitmap representation of the original shape.)

## Bitmap Shapes

To convert another type of shape into a bitmap shape, you use the `GXSetShapeType` function, which is described in detail in the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects*.

Listing 5-9 shows a sample function that defines a figure-eight geometry, encapsulates the geometry in a path shape, sets the pen width of that path to 10, and skews the path around its center by 10% along both the horizontal and vertical axes. Then the sample function converts the path shape into a bitmap shape and draws the bitmap.

---

**Listing 5-9**      Converting a path to a bitmap

```
void ConvertPathToBitmap(void)
{
    gxShape pathToBitmapShape;

    gxRectangle theBounds;

    const long figureEightGeometry[] = {1, /* number of contours */
                                         4, /* number of points */
                                         0xF0000000, /* 1111 ... */
                                         ff(20), ff(20), /* off */
                                         ff(100), ff(100), /* off */
                                         ff(20), ff(100), /* off */
                                         ff(100), ff(20)}; /* off */

    pathToBitmapShape = GXNewPaths((gxPaths *) figureEightGeometry);
    GXSetShapeFill(pathToBitmapShape, gxClosedFrameFill);
    GXSetShapePen(pathToBitmapShape, ff(10));
    GXSkewShape(pathToBitmapShape, fl(.1), fl(.1), ff(60), ff(60));

    GXSetShapeType(pathToBitmapShape, gxBitmapType);
    GXDrawShape(pathToBitmapShape);
    GXDisposeShape(pathToBitmapShape);
}
```

Listing 5-9 uses the `GXSkewShape` function, which is described fully in the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

Figure 5-18 shows the result of this function.

---

**Figure 5-18** A bitmap representation of a path shape



Notice that QuickDraw GX draws the bitmap at 72 pixels per inch.

When converting shapes to bitmap shapes, QuickDraw GX creates a bitmap shape with the smallest pixel image possible to contain the bitmap representation of the original shape. To illustrate, you can draw the bounding rectangle of the skewed figure-eight bitmap by adding to Listing 5-9 the declaration

```
gxRectangle theBounds;
```

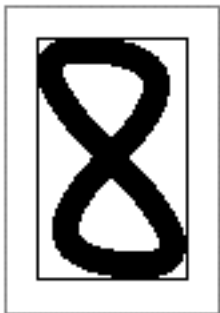
and these two lines of code:

```
GXGetShapeBounds(pathToBitmapShape, 0, &theBounds);  
GXDrawRectangle(&theBounds, gxClosedFrameFill);
```

The resulting bitmap and bounding rectangle are shown in Figure 5-19.

---

**Figure 5-19** A bitmap and its bounding rectangle



## Bitmap Shapes

When QuickDraw GX converts other types of shapes into a bitmap shape, it creates a new bitmap geometry and draws the original shape into the bitmap's pixel image. If the original shape does not cover all of the pixels in the bitmap's pixel image, QuickDraw GX sets the color value of the extra pixels to white. These white pixels may produce unexpected results if you draw the bitmap over a background that includes colors other than white.

For example, the following code adds a background shape to the sample function in Listing 5-9:

```
gxShape backgroundShape;
const gxRectangle backgroundBounds = {ff(20), ff(10),
                                       ff(100), ff(110)};

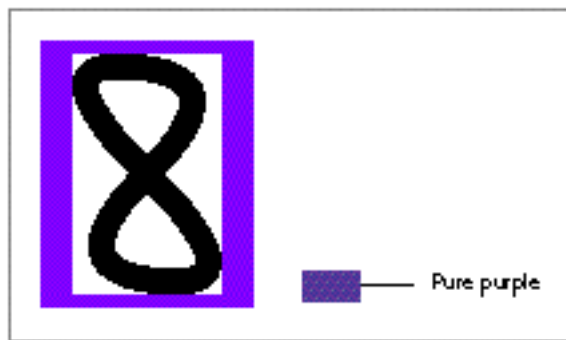
backgroundShape = GXNewRectangle(&backgroundBounds);
SetShapeCommonColor(backgroundShape, purple);
```

If you draw the background before the bitmap, the white pixels of the bitmap cover the corresponding area of the purple rectangle:

```
GXDrawShape(backgroundShape);
GXDrawShape(pathToBitmapShape);
```

The result appears as shown in Figure 5-20. For a color version of this figure, see Plate 6 at the front of this book.

**Figure 5-20** A bitmap drawn over a background

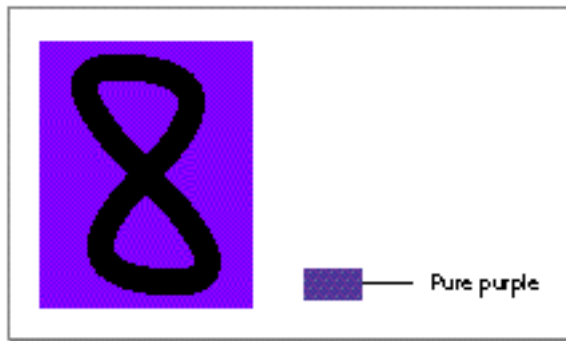


You can set the transfer mode of the bitmap shape to allow the purple to show through the white pixels. For example, you can set the transfer mode of the bitmap to the `gxMinimumMode` transfer mode using this code:

```
SetShapeCommonTransfer(pathToBitmapShape, gxMinimumMode);
GXDrawShape(pathToBitmapShape);
```

The result is shown in Figure 5-21. For a color version of this figure, see Plate 6 at the front of this book.

**Figure 5-21** A bitmap with a transfer mode drawn over a background



Another way to allow the purple rectangle to show through the white areas of this bitmap is to set the clip shape of the bitmap. The next section, “Applying Transformations to Bitmaps,” shows an example of clipping a bitmap.

The examples in this section use colors and the `SetCommonColor` library function, which are available in the color library, and transfer modes and the `SetShapeCommonTransfer` library function, which are available in the transfer mode library.

For more information about the `GXSetShapeType` function, see the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

For information about combining multiple QuickDraw GX shapes into a single bitmap shape, see “Creating Bitmaps Offscreen,” which begins on page 5-45.

## Applying Transformations to Bitmaps

Although bitmap shapes make limited use of their style and ink objects, they make full use of their transform objects. The examples in this section show how you can use the transform object of a bitmap to affect the drawing of that bitmap. The first few sample functions illustrate mapping transformations, and the last sample function illustrates clipping.

## Mapping Bitmap Shapes

---

Since a bitmap geometry contains a pixel image rather than a geometric description, applying mapping transformations to bitmap shapes does not produce the same quality results as applying mapping transformations to geometric shapes. To use as an example, Figure 5-22 shows the path shape converted to a bitmap in Listing 5-9 on page 5-35.

---

**Figure 5-22** A path shape converted to a bitmap shape



You can call the `GXSkewShape` function to undo the skewing of the figure-eight shape:

```
GXSkewShape(pathToBitmapShape, -f1(.1), -f1(.1), ff(60), ff(60));
```

Figure 5-23 shows the results of performing this transformation on the figure-eight bitmap shape.

---

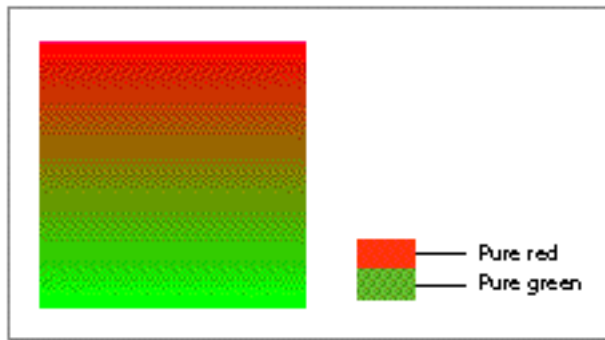
**Figure 5-23** A path shape converted to a bitmap shape and then skewed



## Bitmap Shapes

As Figure 5-23 shows, the quality of the transformed bitmap has degraded due to the skewing. If the `gxMapTransformShape` shape attribute of the bitmap shape is not set, this degradation of quality becomes more pronounced with multiple transformations. For example, consider the color ramp depicted in Figure 5-24. For a color version of this figure, see Plate 4 at the front of this book.

---

**Figure 5-24** A color ramp bitmap


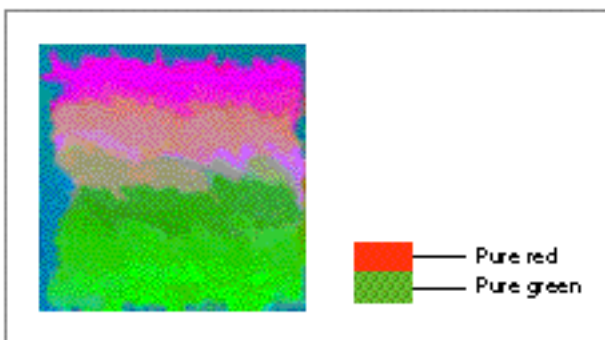
The following lines of code clear the `gxMapTransformShape` shape attribute for this bitmap shape and then rotate the shape 360 times by 1 degree each time:

```
GXSetShapeAttributes(aColorRampBitmapShape,
                    GXGetShapeAttributes(aColorRampBitmapShape)
                    & ~gxMapTransformShape);

for (count = 1; count <= 360; count++)
    GXRotateShape(aColorRampBitmapShape, ff(1), ff(100), ff(100));
```

Enough error is introduced to create an interesting new bitmap, as shown in Figure 5-25. For a color version of this figure, see Plate 5 at the front of this book.

---

**Figure 5-25** A bitmap after multiple transformations




## Bitmap Shapes

However, if you leave the `gxMapTransformShape` shape attribute set, you can apply the same 360 transformations, and the resulting bitmap is identical to the original bitmap. In this case, all of the transformations affect the mapping matrix stored in the bitmap's transform object and not the pixel values of the bitmap's pixel image.

Scaling text provides another example of transformations degrading the quality with which QuickDraw GX draws a shape. As an example, the sample function in Listing 5-10 creates a text shape, draws it, scales it up, and then draws the scaled version. This sample function uses the `GXScaleShape` function, which is described in the chapter "Transform Objects" of *Inside Macintosh: QuickDraw GX Objects*.

---

**Listing 5-10**     Scaling text

```
void ScaleText(void)
{
    gxShape    aTextShape;
    const gxPoint initialLocation = {ff(50), ff(50)};

    aTextShape = GXNewText(9, (unsigned char *) "123456789",
                          &initialLocation) ;
    GXDrawShape(aTextShape);

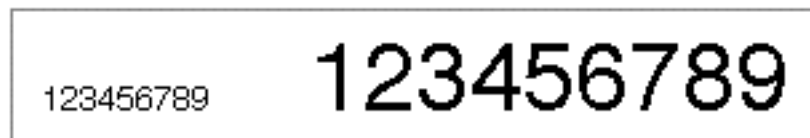
    GXScaleShape(aTextShape, ff(3), ff(3), ff(0), ff(50));

    GXDrawShape(aTextShape);
    GXDisposeShape(aTextShape);
}
```

The result is shown in Figure 5-26.

---

**Figure 5-26**     Scaled text



## Bitmap Shapes

If you convert the text shape to a bitmap shape before scaling it, as in the sample function in Listing 5-11, the result is quite different.

---

**Listing 5-11**    Scaling a bitmap

```
void ScalingABitmap(void)
{
    gxShape aBitmapShape;
    gxPoint initialLocation = {ff(50), ff(50)};

    aBitmapShape = GXNewText(9, (unsigned char *) "123456789",
                             &initialLocation) ;

    GXSetShapeType(aBitmapShape, gxBitmapType);

    GXDrawShape(aBitmapShape);

    GXScaleShape(aBitmapShape, ff(3), ff(3), ff(0), ff(50));

    GXDrawShape(aBitmapShape);
    GXDisposeShape(aBitmapShape);
}
```

Figure 5-27 compares the result of scaling the text shape with the result of scaling the bitmap shape.

---

**Figure 5-27**    Scaled text and a scaled bitmap



## Bitmap Shapes

When scaling the text, QuickDraw GX uses the outline information in the font to draw the best representation of the text at the appropriate size. When scaling the bitmap representation of the text, QuickDraw GX simply scales the bits used to represent the smaller version of the text.

For more information about text shapes, see *Inside Macintosh: QuickDraw GX Typography*.

## Clipping Bitmap Shapes

You can use the transform object of a bitmap shape to clip the bitmap—that is, restrict the area where QuickDraw GX draws the bitmap.

As an example, to apply a circular clip to the color ramp from Figure 5-24 on page 5-40, you start by defining the circular geometry and encapsulating it in a path shape:

```
long theClipGeometry[] = {1, 4, 0xF0000000,
                          ff(50), ff(50),
                          ff(150), ff(50),
                          ff(150), ff(150),
                          ff(50), ff(150)};

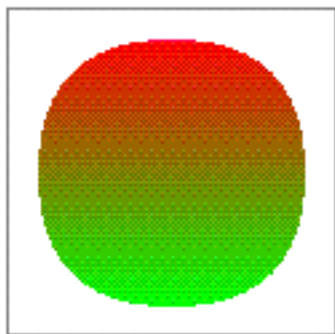
aClipShape = GXNewPaths((gxPaths *) theClipGeometry);
```

Then set the clip property of the bitmap's transform object by using this call to the `GXSetShapeClip` function:

```
GXSetShapeClip(aColorRampBitmapShape, aClipShape);
```

QuickDraw GX draws the resulting bitmap shape as shown in Figure 5-28. For a color version of this figure, see Plate 5 at the front of this book.

**Figure 5-28** A clipped bitmap



For more information about transform objects, mapping transformations, clip shapes, and the `GXSetShapeClip` function, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## Creating Bitmaps With Disk-Based Pixel Images

---

QuickDraw GX allows you to store the pixel image of a bitmap shapes in a disk file. To create this type of bitmap, you specify a predefined constant for the image field of the bitmap's geometry:

```
aBitmapGeometry.image = gxBitmapFileAliasImageValue;
```

The other fields of the geometry you can initialize as you would for other bitmaps:

```
aBitmapGeometry.width = widthOfDiskBasedImage;
aBitmapGeometry.height = heightOfDiskBasedImage;
aBitmapGeometry.rowBytes = rowBytesOfDiskBasedImage;
aBitmapGeometry.pixelSize = pixelSizeOfDiskBasedImage;

aBitmapGeometry.space = colorSpaceOfDiskBasedImage;
aBitmapGeometry.set = colorSetOfDiskBasedImage;
aBitmapGeometry.profile = colorProfileOfDiskBasedImage;
```

You still create the bitmap using the `GXNewBitmap` function:

```
aBitmapShape = GXNewBitmap(&aBitmapGeometry, &initialLocation);
```

You specify the file that contains the pixel image using the bitmap data source alias structure, which is defined by the `gxBitmapDataSourceAlias` data type:

```
struct gxBitmapDataSourceAlias {
    unsigned long fileOffset; /* offset (in bytes) to image */
    unsigned long aliasRecordSize; /* size of alias record */
    unsigned char aliasRecord[gxAnyNumber]; /* alias record */
};
```

To use this data type, you need to declare a variable to hold the structure:

```
gxBitmapDataSourceAlias anAlias;
```

Then, you need to set the three fields of the structure:

- n the `aliasRecord` field should contain a Macintosh Alias Manager alias record specifying the file containing the pixel image
- n the `aliasRecordSize` field should specify the size in bytes of the alias record
- n the `fileOffset` field should specify the offset in bytes from the beginning of the data fork of the file to the first pixel value of the pixel image

## Bitmap Shapes

Once you've created the bitmap data source alias structure, you create a tag object to encapsulate the structure, using the call

```
anAliasTag = GXNewTag(gxBitmapFileAliasTagType, sizeof(anAlias)
                    &anAlias);
```

Then you associate the tag object with the bitmap shape using the call

```
GXSetShapeTags(aBitmapShape, gxBitmapFileAliasTagType,
               1, /* first tag */
               -1, /* replace all tags of same type */
               1, /* insert one new tag */
               &anAliasTag); /* tag to insert */
```

Now the disk-based bitmap is completely initialized. You can use most bitmap-related functions with this bitmap, but there are bitmap-related functions you cannot use. In particular, you cannot call the `GXSetBitmapParts`, `GXSetShapePixel`, `GXNewViewDevice`, or `GXSetViewDeviceBitmap` functions, as these functions would require QuickDraw GX to write to the file.

For more information about alias records, see the chapter “Alias Manager” of *Inside Macintosh: Files*.

For more information about tags and the `GXNewTag` function, see the chapter “Tag Objects” of *Inside Macintosh: QuickDraw GX Objects*. For information about the `GXSetShapeTags` function, see the chapter “Shape Objects” in that book.

## Creating Bitmaps Offscreen

---

The section “Converting Other Types of Shapes to Bitmaps” beginning on page 5-34 describes how you can convert a single QuickDraw GX shape to a bitmap shape. This section shows you how to draw multiple QuickDraw GX shapes to a single bitmap shape.

When you draw a shape, QuickDraw GX does the following:

- n examines the shape's transform object, which contains a view port list
- n examines the view ports in this list, each of which belongs to a view group
- n examines these view groups, which contain view devices
- n decides which view devices the shape actually intersects
- n examines these view devices, each of which contains a bitmap
- n renders the shape into these bitmaps

## Bitmap Shapes

Therefore, to draw shapes into an offscreen bitmap, you need to

- n create a bitmap shape to contain the rendered shapes
- n create a view group to contain a view device
- n create a view device to contain the bitmap shape
- n create a view port that belongs to the view group
- n create a transform to reference the view port
- n associate the transform with the shapes you want to draw offscreen
- n clear the offscreen bitmap
- n draw the shapes

You can find complete information about transforms, view devices, view groups, and view ports in the chapters “Transform Objects” and “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To create the offscreen bitmap, you must define a shape reference for the bitmap shape and create a bitmap shape of the appropriate size:

```
gxShape  aBitmapShape;

aBitmapShape = CreateABitmap(200, 200);
```

Listing 5-12 shows a possible definition for the `CreateABitmap` function. This function creates a black-and-white bitmap of a specified height and width.

---

**Listing 5-12**    Creating a black-and-white bitmap

```
static gxShape CreateABitmap(long height, long width)
{
    gxShape aBitmapShape;
    gxBitmap aBitmapGeometry;
    const gxPoint initialLocation = {ff(0), ff(0)};

    aBitmapGeometry.image = nil;

    aBitmapGeometry.width = width;
    aBitmapGeometry.height = height;
    aBitmapGeometry.rowBytes = 0;
    aBitmapGeometry.pixelSize = 1;

    aBitmapGeometry.space = gxNoSpace;
    aBitmapGeometry.set = nil;
    aBitmapGeometry.profile = nil;
```

## Bitmap Shapes

```

    aShape = GXNewBitmap(&aBitmapGeometry, &initialLocation);

    return(aBitmapShape);
}

```

To create the offscreen view device, view group, and view port objects, you must declare references to them:

```

gxViewGroup offscreenViewGroup;
gxViewDevice offscreenViewDevice;
gxViewPort offscreenViewPort;

```

You create the view group object first:

```
offscreenViewGroup = GXNewViewGroup();
```

Then you can create the view device and view port objects. To create a view device, you must specify both the view group it belongs to and the bitmap it uses when rendering shapes:

```

offscreenViewDevice = GXNewViewDevice(offscreenViewGroup,
                                       aBitmapShape);

```

To create a view port, you need only specify the view group to which it belongs:

```
offscreenViewPort = GXNewViewPort(offscreenViewGroup);
```

To draw shapes to this offscreen view port, you need to create a new transform object. First, you must declare a reference to a transform object:

```
gxTransform offscreenTransform;
```

Then you can create it and set its view port list to contain the offscreen view port:

```

offscreenTransform = GXNewTransform();
GXSetTransformViewPorts(offscreenTransform, 1,
                        &offscreenViewPort);

```

Now you're ready to draw shapes offscreen. The first shape that you draw is a simple white rectangle, and drawing it initializes the pixels in the offscreen bitmap:

```

gxShape aRectangleShape;
gxRectangle boundsRectangle = {ff(0), ff(0), ff(200), ff(200)};

aRectangleShape = GXNewRectangle(&boundsRectangle);

SetShapeCommonColor(aRectangleShape, gxWhite);

```

## Bitmap Shapes

To draw this white rectangle to the offscreen bitmap, you must set its transform object to be the offscreen transform object:

```
GXSetShapeTransform(aRectangleShape, offscreenTransform);
```

Then you draw and dispose of the shape:

```
GXDrawShape(aRectangleShape);
GXDisposeShape(aRectangleShape);
```

Since the rectangle shape references the offscreen transform object, QuickDraw GX draws the white rectangle into the offscreen bitmap.

Because the offscreen bitmap is now initialized, you can draw other shapes to it. The following code demonstrates how to create a line shape and draw it to the offscreen bitmap:

```
gxShape aLineShape;
gxLine lineGeometry = {ff(40), ff(40), ff(160), ff(160)};

aLineShape = GXNewLine(&lineGeometry);
GXSetShapePen(aLineShape, ff(50));
GXSetShapeTransform(aLineShape, offscreenTransform);
GXDrawShape(aLineShape);
GXDisposeShape(aLineShape);
```

As another example, the following code demonstrates how to create a text shape and draw it to the offscreen bitmap:

```
gxShape aTextShape;
gxPoint textLocation = {ff(70), ff(100)};
gxPoint textCenter;

aTextShape = GXNewText(9, (unsigned char *) "123456789",
                        &textLocation);
GXGetShapeCenter(aTextShape, 0, &textCenter);
GXScaleShape(aTextShape, ff(3), ff(3),
             textCenter.x, textCenter.y);
SetShapeCommonTransfer(aTextShape, gxXorMode);

GXSetShapeTransform(aTextShape, offscreenTransform);
GXDrawShape(aTextShape);
GXDisposeShape(aTextShape);
```

This code segment uses the `SetShapeCommonTransfer` library function, which is available in the transfer mode library.



## Bitmap Shapes

Finally, to transfer the offscreen bitmap to the screen, you need only draw the bitmap:

```
GXDrawShape(aBitmapShape);
```

When drawing the offscreen bitmap, QuickDraw GX uses the information in the transform object of the offscreen bitmap shape. This example uses the `GXNewBitmap` function to create the offscreen bitmap, and so it references the same transform object as the default bitmap shape. The transform of the default bitmap references the default view port, as described in the chapter “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*. Since the default view port is typically on screen, drawing the offscreen bitmap effectively transfers it to the screen.

Listing 5-13 shows the complete sample function to create an offscreen bitmap, draw shapes to it, and copy it to the screen.

---

**Listing 5-13**    Creating an offscreen bitmap

```
void CreateOffscreenBitmap(void)
{
    gxShape  aBitmapShape, aRectangleShape, aLineShape, aTextShape;

    gxRectangle boundsRectangle = {ff(0), ff(0), ff(200), ff(200)};
    gxLine lineGeometry = {ff(40), ff(40), ff(160), ff(160)};
    gxPoint textLocation = {ff(70), ff(100)};
    gxPoint textCenter;

    /* declare view group, and so forth. */
    aBitmapShape = CreateABitmap(200, 200);

    offscreenViewGroup = GXNewViewGroup();
    offscreenViewDevice = GXNewViewDevice(offscreenViewGroup,
                                           aBitmapShape);
    offscreenViewPort = GXNewViewPort(offscreenViewGroup);
    offscreenTransform = GXNewTransform();
    GXSetTransformViewPorts(offscreenTransform, 1,
                           &offscreenViewPort);

    /* draw white rectangle to clear bitmap */
    aRectangleShape = GXNewRectangle(&boundsRectangle);
    GXSetShapeTransform(aRectangleShape, offscreenTransform);
    SetShapeCommonColor(aRectangleShape, gxWhite);
    GXDrawShape(aRectangleShape);
    GXDisposeShape(aRectangleShape);
```

## Bitmap Shapes

```

    /* draw thick diagonal line offscreen */
    aLineShape = GXNewLine(&lineGeometry);
    GXSetShapePen(aLineShape, ff(50));
    GXSetShapeTransform(aLineShape, offscreenTransform);
    GXDrawShape(aLineShape);
    GXDisposeShape(aLineShape);

    /* draw text offscreen */
    aTextShape = GXNewText(9, (unsigned char *) "123456789",
                          &textLocation);
    GXGetShapeCenter(aTextShape, 0, &textCenter);
    GXScaleShape(aTextShape, ff(3), ff(3), textCenter.x,
                textCenter.y);
    SetShapeCommonTransfer(aTextShape, gxXorMode);

    GXSetShapeTransform(aTextShape, offscreenTransform);
    GXDrawShape(aTextShape);
    GXDisposeShape(aTextShape);

    /* transfer bitmap to screen */
    GXDrawShape(aBitmapShape);
    GXDisposeShape(aBitmapShape);

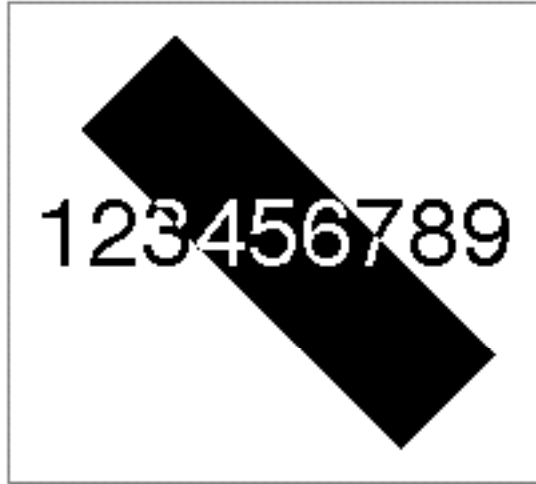
    GXDisposeTransform(offscreenTransform);
    GXDisposeViewGroup(offscreenViewGroup);
}

```

Figure 5-29 shows the result of this function.

---

**Figure 5-29** Multiple shapes drawn to a bitmap



The offscreen library provided with QuickDraw GX contains some utilities that simplify the creation of offscreen bitmaps. This library defines the `offscreen` structure, which contains a reference to a transform, view port, view device, and view group. Listing 5-14 shows how to use the offscreen library to create the bitmap shown in Figure 5-29.

---

**Listing 5-14** Creating an offscreen bitmap using the offscreen library

```
void CreateOffscreenBitmap(void)
{
    shape    aBitmapShape, aRectangleShape, aLineShape, aTextShape;

    offscreen anOffscreen;

    const gxRectangle boundsRectangle = {ff(0), ff(0),
                                          ff(200), ff(200)};
    const gxLine lineGeometry = {ff(40), ff(40),
                                  ff(160), ff(160)};
    const gxPoint textLocation = {ff(70), ff(100)};
    gxPoint textCenter;

    aBitmapShape = CreateABitmap(200, 200);
```

## Bitmap Shapes

```

/* create all offscreen-related objects */
CreateOffscreen(&anOffscreen, aBitmapShape);

aRectangleShape = GXNewRectangle(&boundsRectangle);
GXSetShapeTransform(aRectangleShape, anOffscreen.xform);
SetShapeCommonColor(aRectangleShape, gxWhite);
GXDrawShape(aRectangleShape);
GXDisposeShape(aRectangleShape);

aLineShape = GXNewLine(&lineGeometry);
GXSetShapePen(aLineShape, ff(50));
GXSetShapeTransform(aLineShape, anOffscreen.xform);
GXDrawShape(aLineShape);
GXDisposeShape(aLineShape);

aTextShape = GXNewText(9, (unsigned char *) "123456789",
                        &textLocation);
GXGetShapeCenter(aTextShape, 0, &textCenter);
GXScaleShape(aTextShape, ff(3), ff(3), textCenter.x,
             textCenter.y);
SetShapeCommonTransfer(aTextShape, gxXorMode);

GXSetShapeTransform(aTextShape, anOffscreen.xform);
GXDrawShape(aTextShape);
GXDisposeShape(aTextShape);

GXDrawShape(aBitmapShape);
GXDisposeShape(aBitmapShape);

/* dispose of all offscreen-related objects */
DisposeOffscreen(&anOffscreen);

GXDrawShape(aBitmapShape);
GXDisposeShape(aBitmapShape);
}

```

## Editing Part of a Bitmap

---

QuickDraw GX provides two functions that allow you to manipulate part of a bitmap. The `GXGetBitmapParts` function copies a rectangular subsection from one bitmap to a new bitmap, and the `GXSetBitmapParts` function replaces a rectangular subsection of one bitmap with another bitmap.

To extract part of a bitmap shape, you need to declare a reference to a new bitmap shape to hold the extracted part:

```
gxShape extractedBitmap;
```

You also need to specify what part of the bitmap to extract. QuickDraw GX provides the `gxLongRectangle` structure for this purpose:

```
gxLongRectangle extractedBounds = {70, 70, 125, 125};
```

You can then use the `GXGetBitmapParts` function to extract the specified section. For example, the following call extracts from the bitmap referenced by the `aBitmapShape` variable the section starting at 70 pixels over and 70 pixels down and ending at 125 pixels over and 125 pixels down.

```
extractedBitmap = GXGetBitmapParts(aBitmapShape,
                                   &extractedBounds);
```

Applying this function call to the bitmap shown in Figure 5-29 results in the bitmap shown in Figure 5-30.

**Figure 5-30** An extracted bitmap



You can use the `GXSetBitmapParts` function to replace a section of one bitmap with the contents of another bitmap.

For example, you might create a small, square bitmap containing all black pixels:

```
gxShape insertionBitmap;
gxRectangle insertionGeometry = {ff(0), ff(0), ff(100), ff(100)};

insertionBitmap = GXNewRectangle(&insertionGeometry);
GXSetShapeType(insertionBitmap, gxBitmapType);
```

## Bitmap Shapes

Then you can insert that bitmap into the bitmap from Figure 5-29 by specifying where it should be inserted with the declaration

```
gxLongRectangle whereToInsert = {70, 70, 125, 125};
```

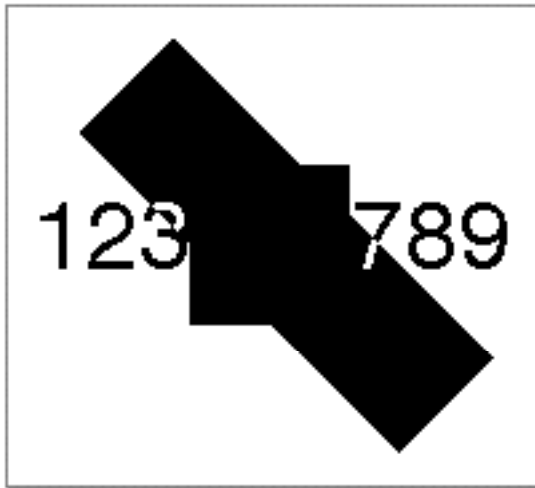
and then inserting it with this call to the `GXSetBitmapParts` function:

```
GXSetBitmapParts(aBitmapShape, &whereToInsert, insertionBitmap);
```

Notice that the `insertionBitmap` shape is larger than the `whereToInsert` rectangle. QuickDraw GX only inserts as much of the `insertionBitmap` shape as fits in the `whereToInsert` rectangle, starting with the upper-left corner of the `insertionBitmap` shape.

The resulting bitmap is shown in Figure 5-31.

**Figure 5-31** An edited bitmap



For more information about the `GXGetBitmapParts` and the `GXSetBitmapParts` functions, see page 5-74 and page 5-75, respectively.

## Applying Functions Described Elsewhere to Bitmap Shapes

QuickDraw GX provides only a small number of functions that apply exclusively to bitmaps. However, most of the QuickDraw GX functions that apply to other types of shapes can also be applied to bitmap shapes.

## Bitmap Shapes

The next seven sections discuss how functions described elsewhere operate on bitmaps. These sections are as follows:

- n “Functions That Post Errors or Warnings When Applied to Bitmap Shapes” on page 5-55, which lists functions that you can apply to other types of shapes but not to bitmap shapes
- n “Shape-Related Functions Applicable to Bitmap Shapes” on page 5-56, which lists functions that operate on bitmap shape objects
- n “Geometric Operations Applicable to Bitmap Shapes” on page 5-58, which lists the few geometric operation functions that you can apply to bitmap geometries
- n “Style-Related Functions Applicable to Bitmap Shapes” on page 5-59, which lists the few style-related functions that affect the drawing of bitmaps
- n “Ink-Related Functions Applicable to Bitmap Shapes” on page 5-59, which lists the functions that manipulate on the transfer mode of a bitmap shape’s ink object
- n “Transform-Related Functions Applicable to Bitmap Shapes” on page 5-59, which discusses the functions that allow you to map and clip a bitmap as well as set its hit-test parameters and its view port list
- n “View-Related Functions Applicable to Bitmap Shapes” on page 5-61, which lists the functions that allow you to associate a bitmap shape with a view device object

## Functions That Post Errors or Warnings When Applied to Bitmap Shapes

---

Some QuickDraw GX functions that operate on other types of shapes only post an error or a warning if you try to apply them to a bitmap shape.

For example, the shape-editing functions listed in Table 5-1 operate on the geometric shape types, but not on bitmap shapes. These functions are described in Chapter 2, “Geometric Shapes,” in this book.

**Table 5-1** Shape-editing functions that post errors or warnings when applied to bitmaps

Function name	Error or warning posted
<code>GXGetShapeParts</code>	<code>shape_operator_may_not_be_a_bitmap</code>
<code>GXSetShapeParts</code>	<code>shape_operator_may_not_be_a_bitmap</code>

Although you cannot apply the functions listed in Table 5-1 to a bitmap shape, you can use the `GXGetBitmapParts` and `GXSetBitmapParts` functions to edit sections of a bitmap. These functions are described in “Editing Bitmaps” beginning on page 5-71.

## Bitmap Shapes

There are also a number of geometric operations that you cannot apply to bitmap shapes. Table 5-2 lists these functions, which are described in Chapter 4, “Geometric Operations,” in this book.

**Table 5-2** Geometric operations that post errors or warnings when applied to bitmaps

Function name	Error or warning posted
GXBreakShape	graphic_type_does_not_contain_points
GXContainsShape	shape_operator_may_not_be_a_bitmap
GXDifferenceShape	shape_operator_may_not_be_a_bitmap
GXExcludeShape	shape_operator_may_not_be_a_bitmap
GXGetShapeCenter	illegal_type_for_shape
GXGetShapeDirection	graphic_type_does_not_have_multiple_contours
GXGetShapeLength	shape_does_not_have_length
GXInsetShape	graphic_type_cannot_be_inset
GXIntersectShape	shape_operator_may_not_be_a_bitmap
GXInvertShape	shape_cannot_be_inverted
GXReduceShape	graphic_type_cannot_be_reduced
GXReverseDifferenceShape	shape_operator_may_not_be_a_bitmap
GXReverseShape	contour_out_of_range
GXShapeLengthToPoint	shape_does_not_have_length
GXTouchesShape	shape_operator_may_not_be_a_bitmap
GXUnionShape	shape_operator_may_not_be_a_bitmap

Most of these geometric operations do not apply to bitmap shapes because the geometry of a bitmap is substantially different from the geometry of a geometric shape.

You can apply a few of the geometric operations to bitmaps, however. These functions are discussed in “Geometric Operations Applicable to Bitmap Shapes” beginning on page 5-58.

## Shape-Related Functions Applicable to Bitmap Shapes

You can apply all of the functions described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects* to bitmap shapes. These functions allow you to

- n manipulate the shape object that represents the bitmap shape—for example, you can copy, clone, cache, compare, and dispose of the bitmap shape
- n set the geometry, shape type, shape fill, and shape attributes of the bitmap shape



## Bitmap Shapes

- n change the style, ink, and transform objects that are associated with the bitmap shape
- n manipulate the tags and owner count of the bitmap shape

Table 5-3 gives important bitmap-related information for a subset of the functions from the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*. Functions described in that chapter that do not appear in this list exhibit the same behavior when applied to bitmap shapes as they do when applied to other types of shapes.

**Table 5-3** Shape-related functions that exhibit special behavior when applied to bitmaps

Function name	Action taken
GXChangedShape	Notifies QuickDraw GX that you have directly edited the geometry of the bitmap (using the <code>GXGetShapeStructure</code> and <code>GXLockShape</code> functions). You should call this function when you directly edit any field of a bitmap geometry structure or when you edit the pixel values of a bitmap’s pixel image.
GXCopyToShape	Makes a copy of the bitmap shape, but does not copy the pixel image. Instead, the new bitmap shape references the same pixel image.
GXCopyDeepToShape	Makes a copy of the bitmap shape, including a complete copy of the bitmap’s pixel image. The copy of the pixel image is allocated in QuickDraw GX memory, regardless of where the original image was allocated.
GXEqualShape	Determines if two bitmap shapes are equal—that is, their bitmap position, height, width, color space, color set, and color profile fields are equal, and their pixel images contain the same pixel values.
GXGetShapeSize	Determines the amount of memory currently used by the bitmap shape, including the amount of QuickDraw GX memory currently used by the pixel image of the bitmap.
GXGetDefaultShape	Returns a reference to the default bitmap shape. The default bitmap shape has 1 bit per pixel, 0 width, and 0 height.
GXGetShapeFill	Returns the shape fill of the shape. The shape fill for bitmap shapes is always even-odd fill or no fill.
GXGetShapeStructure	Returns a pointer to the geometry of the bitmap shape. You can use this function to determine the address of the pixel image, even if it is allocated in QuickDraw GX memory.
GXLockShape	Loads the bitmap shape into memory and locks its geometry into a fixed memory location. If the pixel image is allocated in QuickDraw GX memory, it is loaded and locked as well.
GXNewShape	Creates a bitmap shape with 0 width, 0 height, 32 bits per pixel and <code>rgb32space</code> color space.
GXSetShapeFill	Sets the shape fill of the shape. You must always set the shape fill of a bitmap shape to even-odd fill or no fill.
GXSetShapeType	Changes the shape type of the bitmap shape and converts the shape fill and geometry as appropriate.

## Geometric Operations Applicable to Bitmap Shapes

---

Most geometric operations post errors or warnings when applied to bitmap shapes, as described in “Functions That Post Errors or Warnings When Applied to Bitmap Shapes” on page 5-55.

You can, however, apply the remainder of the functions described in Chapter 4, “Geometric Operations,” to bitmap shapes. Table 5-4 gives important bitmap-related information for a subset of these functions. The remainder of the geometric operations exhibit the same behavior when applied to bitmap shapes as they do when applied to other types of shapes.

**Table 5-4** Geometric operations that exhibit special behavior when applied to bitmaps

---

Function name	Action taken
<code>GXGetShapeArea</code>	Returns bitmap width multiplied by bitmap height.
<code>GXPrimitiveShape</code>	Applies <code>sourceGridStyle</code> attribute to bitmap position.
<code>GXSimplifyShape</code>	Reduces the pixel size of the bitmap if the bitmap uses a limited number of colors.
<code>GXSetShapeBounds</code>	If the <code>gxMapTransformShape</code> shape attribute is set, this function changes the transform mapping of the bitmap; otherwise, it changes the bitmap height, width, and location, and creates a new, scaled version of the bit image to fit in the new bounding rectangle.

## Style-Related Functions Applicable to Bitmap Shapes

---

As discussed in “Bitmap Styles and Inks” on page 5-8, bitmap shapes make limited use of their style objects. Although you can apply to a bitmap shape any of the functions described in Chapter 3, “Geometric Styles,” only the `GXSetShapeStyleAttributes` function affects the drawing of the bitmap. While you can use this function to set or clear any of a bitmap’s style attributes, QuickDraw GX considers only the `gxSourceGridStyle` style attribute and the `gxDeviceGridStyle` style attribute when drawing bitmaps; other attributes are ignored.

You may use the other style-related functions (such as `GXSetShapePen`, `GXSetShapeDash`, and so on) to set the other properties of a bitmap’s style object, and you may use the corresponding functions (`GXGetShapePen`, `GXGetShapeDash`, and so on) to examine these properties. However, QuickDraw GX ignores these properties when drawing a bitmap.

## Ink-Related Functions Applicable to Bitmap Shapes

---

Since bitmap shapes contain their own color information in their geometries, QuickDraw GX does not use the color property of the ink object when drawing a bitmap. However, QuickDraw GX does consider the transfer mode of the ink object and applies it to each pixel when drawing a bitmap. You can use the `GXSetShapeTransfer` function, which is described in the chapter “Ink Objects” in *Inside Macintosh: QuickDraw GX Objects*, to assign a transfer mode to a bitmap shape.

You may also use the `GXSetShapeColor` function to set the color property of a bitmap’s ink object and use the `GXGetShapeColor` function to examine this property. However, QuickDraw GX ignores this property when drawing a bitmap.

## Transform-Related Functions Applicable to Bitmap Shapes

---

Although bitmap shapes do not make full use of their style and ink objects, they do make full use of their transform objects. You can apply all of the shape-related functions that are described in the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects* to bitmap shapes.

## Bitmap Shapes

Table 5-5 gives important bitmap-related information for a subset of the functions from that chapter. Functions described in that chapter that do not appear in this list exhibit the same behavior when applied to bitmap shapes as they do when applied to other types of shapes.

**Table 5-5** Transform-related functions that exhibit special behavior when applied to bitmaps

Function name	Action taken
<code>GXGetShapeHitTest</code>	Returns the hit-test parameters associated with the bitmap shape's transform. QuickDraw GX hit-tests bitmaps using only the <code>boundsPart</code> shape part.
<code>GXMapShape</code>	Multiplies the mapping associated with the transform object of the bitmap shape (if the <code>gxMapTransformShape</code> shape attribute of the bitmap shape is set) by a mapping matrix, or applies the mapping directly to the geometry of the bitmap (if the <code>gxMapTransformShape</code> attribute is not set). Depending on the mapping, this function may also change the clip shape of the bitmap.
<code>GXMoveShape</code>	Moves the bitmap by a specified distance. This function can affect the mapping of the bitmap's transform or the geometry of the bitmap itself, depending on the value of the <code>gxMapTransformShape</code> shape attribute of the bitmap shape.
<code>GXMoveShapeTo</code>	Moves the bitmap to a specified position. This function can affect the mapping of the bitmap's transform or the geometry of the bitmap itself, depending on the value of the <code>gxMapTransformShape</code> shape attribute of the bitmap shape.
<code>GXRotateShape</code>	Rotates the bitmap. This function can affect the mapping of the bitmap's transform or the geometry of the bitmap itself, depending on the value of the <code>gxMapTransformShape</code> shape attribute of the bitmap shape. This function can also affect the clip shape of the bitmap.
<code>GXScaleShape</code>	Scales the bitmap. This function can affect the mapping of the bitmap's transform or the geometry of the bitmap itself, depending on the value of the <code>gxMapTransformShape</code> shape attribute of the bitmap shape. This function can also affect the clip shape of the bitmap.
<code>GXSkewShape</code>	Skews the bitmap. This function can affect the mapping of the bitmap's transform or the geometry of the bitmap itself, depending on the value of the <code>gxMapTransformShape</code> shape attribute of the bitmap shape. This function can also affect the clip shape of the bitmap.
<code>GXSetShapeClip</code>	Assigns a clip shape to the transform object associated with the bitmap shape.
<code>GXSetShapeHitTest</code>	Assigns hit-test parameters to the transform object associated with the bitmap shape. QuickDraw GX only hit-tests bitmaps using the <code>boundsPart</code> shape part.
<code>GXSetShapeMapping</code>	Changes the mapping associated with the transform object of the bitmap shape (if the <code>gxMapTransformShape</code> shape attribute of the bitmap shape is set) or applies the mapping directly to the geometry of the bitmap (if the <code>gxMapTransformShape</code> attribute is not set). Depending on the mapping, this function may also change the clip shape of the bitmap.

## View-Related Functions Applicable to Bitmap Shapes

---

As described in “Bitmaps and View Devices” beginning on page 5-12, view device objects use bitmaps to store rendered shape images. Table 5-6 lists the function that allows you to determine the bitmap assigned to a view device and the function that allows you to change the bitmap of a view device. Both of these functions are described in detail in the chapter “View-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

**Table 5-6** View-related functions that can be applied to bitmaps

Function name	Action taken
GXGetViewDeviceBitmap	Returns the bitmap shape associated with a view device object.
GXSetViewDeviceBitmap	Assigns a bitmap shape to a view device. You can use this function to create offscreen bitmaps, which are discussed in “Bitmaps and View Devices” beginning on page 5-12.

## Bitmap Shapes Reference

---

This section describes the data types and functions you use to create and manipulate bitmap shapes. The first subsection, “Constants and Data Types,” shows the definitions of the data types related to bitmap shapes. The section “Functions,” beginning on page 5-65, gives a complete reference for the functions specific to bitmaps. In addition to the functions described in this section, you can apply many functions described elsewhere to bitmap shapes. See the section “Applying Functions Described Elsewhere to Bitmap Shapes,” which begins on page 5-54, for more details.

### Constants and Data Types

---

This section describes the data structures you use to create and manipulate bitmaps.

You use the `gxBitmap` structure to specify information about a bitmap geometry and information about how QuickDraw GX should create a bitmap shape. You also use this data structure to specify color information for bitmaps. A complete discussion of the QuickDraw GX color architecture appears in the chapter “Color and Color-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

You use the `gxLongRectangle` structure to specify a rectangular subsection of a bitmap pixel image when editing bitmap parts.

This section also discusses the constants and data types you use when creating disk-based bitmaps.

## The Bitmap Geometry Structure

---

The `gxBitmap` structure specifies the geometry of a bitmap shape. You can use this data structure when creating bitmap shapes with the `GXNewBitmap` function, when altering bitmap shapes with the `GXGetBitmap` and `GXSetBitmap` functions, and when directly editing bitmap shapes with the `GXGetShapeStructure` function.

The `gxBitmap` structure is defined as follows:

```
typedef struct {
    char        *image;
    long         width;
    long         height;
    long         rowBytes;
    long         pixelSize;
    gxColorSpace space;
    gxColorSet   set;
    gxColorProfile profile;
} gxBitmap;
```

### Field descriptions

<code>image</code>	A pointer to the pixel image. When creating a bitmap, you can specify <code>nil</code> for this field to indicate that QuickDraw GX should allocate memory for the pixel image of the bitmap.
<code>width</code>	The width of the bitmap in pixels.
<code>height</code>	The height of the bitmap in pixels.
<code>rowBytes</code>	The number of bytes of the pixel image corresponding to each row of the bitmap. This value must be a positive even number.
<code>pixelSize</code>	The number of bits representing a single pixel in the pixel image. This value must be 1, 2, 4, 8, 16, or 32.
<code>space</code>	The color space that QuickDraw GX uses when interpreting the pixel values in the pixel image. When creating a bitmap, you may specify the <code>gxNoSpace</code> constant for this field to indicate that QuickDraw GX should choose a color space for you. If the value of the <code>pixelSize</code> field is 32, QuickDraw GX uses the value <code>gxRGB32Space</code> ; if the pixel size is 16, QuickDraw GX uses <code>gxRGB16Space</code> ; if the pixel size is 8 or less, QuickDraw GX uses <code>gxIndexedSpace</code> and creates the default color set for the pixel size, which is usually a grayscale color set.

## Bitmap Shapes

set	The color set that QuickDraw GX uses when interpreting the pixel values of the pixel image. If the <code>space</code> field contains the value <code>gxIndexedSpace</code> , QuickDraw GX interprets the pixel values in the pixel image as indexes to this color set. If the bitmap's color space is not <code>gxIndexedSpace</code> , this field should be <code>nil</code> .
profile	The color matching information about the device on which the bitmap was created. You may provide a reference to a color profile object, or you may set the value of this field to <code>nil</code> .

**Implementation Note**

Version 1.0 of QuickDraw GX limits the bitmap width and the bitmap height to 32,767. <sup>u</sup>

When creating a bitmap, you can allocate the memory for the pixel image of the bitmap yourself and store a pointer to it in the `image` field, or you can set the `image` field to `nil`, which indicates that QuickDraw GX should allocate the pixel image memory.

If you create the pixel image for a bitmap, you must pad the end of each row of the pixel image so that each row contains an even number of bytes. You must store the number of bytes per row in the `rowBytes` field, unless you are creating a bitmap with a pixel image in QuickDraw GX memory, in which case you want to set this field to 0.

If you set the `image` field to `nil` when creating a bitmap, QuickDraw GX does two things:

- n creates an uninitialized pixel image based on the bitmap width and height you specify in the `width` and `height` fields and the pixel size you specify in the `pixelSize` field
- n determines an appropriate `rowBytes` value

If you want to create a bitmap with a disk-based pixel image, you should specify the `gxBitmapFileAliasImageValue` constant for the `image` field.

If you specify the `gxNoSpace` constant for the `space` field, QuickDraw GX chooses an appropriate color space for you, based on the value of the `pixelSize` field.

If you specify the color space yourself, you must be sure the pixel size of that color space matches the value you indicate in the `pixelSize` field.

For a discussion of pixel images, bitmap width, bitmap height, and pixel size, see “Bitmap Geometries” beginning on page 5-5. For a detailed discussion of color spaces, color sets, and color profiles, see the chapter “Color and Color-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

For examples of creating `gxBitmap` structures and bitmap shapes, see “Creating and Drawing Bitmaps” beginning on page 5-15.

## The Long Rectangle Structure

---

The `gxLongRectangle` structure allows you to specify a rectangular subsection of the pixel image of a bitmap shape. It differs from the `gxRectangle` structure, described in the chapter “Geometric Shapes” in this book, in that the coordinates of a `gxLongRectangle` structure have no fractional part.

```
struct gxLongRectangle {
    long    left;
    long    top;
    long    right;
    long    bottom;
};
```

### Field descriptions

<code>left</code>	The left side of the rectangle in number of pixels.
<code>top</code>	The top of the rectangle in number of pixels.
<code>right</code>	The right side of the rectangle in number of pixels.
<code>bottom</code>	The bottom of the rectangle in number of pixels.

You use the `gxLongRectangle` structure when editing parts of a bitmap, as discussed in “Editing Part of a Bitmap” beginning on page 5-53.

## Constants For Bitmaps With Disk-Based Pixel Images

---

QuickDraw GX provides two constants for you to use when creating bitmaps with disk-based pixel images.

```
#define  gxBitmapFileAliasImageValue  0x00000001

#define  gxBitmapFileAliasTagType     'bfil'
```

You indicate that a bitmap uses a disk-based pixel image by setting the bitmap geometry’s `image` field to the `gxBitmapFileAliasImageValue` constant. You specify which file contains the pixel image in a bitmap data source alias structure, which you attach to the bitmap using a tag with the `gxBitmapFileAliasTagType` tag type.

For an example of a bitmap with a disk-based pixel image, see “Creating Bitmaps With Disk-Based Pixel Images” beginning on page 5-44.



## Bitmap Data Source Alias Structure

---

QuickDraw GX provides the bitmap data source alias structure to allow you to specify file information for disk-based pixel images.

```
struct gxBitmapDataSourceAlias {
    unsigned long fileOffset;
    unsigned long aliasRecordSize;
    unsigned char aliasRecord[gxAnyNumber];
};
```

### Field descriptions

<code>fileOffset</code>	The offset in bytes from the beginning of the file to the first pixel value of the pixel image.
<code>aliasRecordSize</code>	The size in bytes of the alias record.
<code>aliasRecord</code>	A Macintosh Alias Manager alias record specifying the file containing the pixel image.

For an example of a bitmap with a disk-based pixel image, see “Creating Bitmaps With Disk-Based Pixel Images” beginning on page 5-44.

## Functions

---

This section describes the functions provided by QuickDraw GX specifically for creating and manipulating bitmap shapes. With the functions described in this section, you can

- n create a new bitmap shape
- n determine and replace the geometry of a bitmap shape
- n edit a single pixel of a bitmap
- n examine or replace a rectangular subsection of a bitmap

The section “Applying Functions Described Elsewhere to Bitmap Shapes,” which begins on page 5-54, contains information about other QuickDraw GX functions that you can apply to bitmap shapes.

## Creating Bitmaps

---

This section describes the function you use to create new bitmap shapes.

The `GXNewBitmap` function requires that you specify information about the bitmap in a `gxBitmap` structure, and the function encapsulates that information in a new bitmap shape.

## GXNewBitmap

---

You can use the `GXNewBitmap` function to create a new bitmap shape.

```
gxShape GXNewBitmap(const gxBitmap *data,
                    const gxPoint *position);
```

<code>data</code>	A pointer to a <code>gxBitmap</code> bitmap structure that specifies information about the bitmap shape you want to create.
<code>position</code>	A pointer to a <code>gxPoint</code> structure that indicates the initial position of the upper-right corner of the bitmap. You may set this parameter to <code>nil</code> to indicate (0.0, 0.0).

*function result* A reference to the newly created bitmap shape.

### DESCRIPTION

The `GXNewBitmap` function creates a new bitmap shape and returns a reference to that shape as its function result.

You specify the initial position of the new bitmap in the `position` parameter, and you specify the rest of the bitmap geometry by creating a `gxBitmap` structure and passing a pointer to it in the `data` parameter.

You must provide values for the `width`, `height`, and `pixelSize` fields of the `gxBitmap` structure.

#### Implementation Note

Version 1.0 of QuickDraw GX limits the bitmap width and the bitmap height to 32,767. <sup>u</sup>

You may specify the pixel image in the `image` field of the bitmap geometry structure, or you may set this field to `nil`, in which case QuickDraw GX allocates memory for the pixel image based on the requested width, height, and pixel size. If you supply the pixel image, you must also supply an appropriate value in the `rowBytes` field of the bitmap geometry structure. If QuickDraw GX allocates the pixel image, you should initialize the `rowBytes` field to 0.

You may indicate a color space for the bitmap in the `space` field of the bitmap geometry structure, but the pixel size of that color space must match the pixel size you specify in the `pixelSize` field. If you specify the `gxNoSpace` constant for the `space` field, QuickDraw GX chooses a color space for you:

- <sup>n</sup> If you indicate in the `pixelSize` field a pixel size of 16 or 32, QuickDraw GX chooses the `gxRGB16Space` color space or the `gxRGB32Space` color space, respectively.
- <sup>n</sup> If you indicate in the `pixelSize` field a pixel size of 1, 2, 4, or 8, QuickDraw GX chooses the `gxIndexedSpace` color space, and creates a default color set of the appropriate size.

## Bitmap Shapes

If you indicate the `gxIndexedSpace` color space for the `space` field, you must provide a color set in the `set` field.

In the `profile` field, you may provide a reference to a color profile describing the color matching information for the device on which the bitmap was created, or you may set this field to `nil`.

## SPECIAL CONSIDERATIONS

If no error results, the `GXNewBitmap` function creates a bitmap shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about creating and disposing of shapes.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>size_of_bitmap_exceeds_implementation_limit</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>invalid_pixelSize</code>	(debugging version)
<code>bitmap_height_negative</code>	(debugging version)
<code>bitmap_width_negative</code>	(debugging version)
<code>bitmap_height_negative</code>	(debugging version)
<code>bitmap_rowBytes_negative</code>	(debugging version)
<code>bitmap_rowBytes_too_small</code>	(debugging version)
<code>bitmap_rowBytes_not_aligned</code>	(debugging version)
<code>bitmap_ptr_not_aligned</code>	(debugging version)
<code>bitmap_rowBytes_must_be_specified_for_user_image_buffer</code>	(debugging version)
<code>colorSpace_out_of_range</code>	(debugging version)

**Warnings**

<code>shape_access_not_allowed</code>	(debugging version)
---------------------------------------	---------------------

## SEE ALSO

For examples using this function, see “Creating and Drawing Bitmaps” beginning on page 5-15.

For information about the `gxBitmap` structure, see “The Bitmap Geometry Structure” beginning on page 5-62.

For information about bitmap width, height, and pixel size, see “Bitmap Geometries” beginning on page 5-5.

For information about disposing of bitmap shapes, see the description of the `GXDisposeShape` function, which is in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

For a complete discussion of the QuickDraw GX color architecture, see the chapter “Color and Color-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## Getting and Setting Bitmap Geometries

---

This section describes the functions you can use to examine or replace the entire geometry of a bitmap shape.

The `GXGetBitmap` function copies the information from the geometry of a bitmap shape into a `gxBitmap` data structure.

The `GXSetBitmap` function replaces the geometry of a bitmap shape with information you provide in a `gxBitmap` structure.

## GXGetBitmap

---

You can use the `GXGetBitmap` function to obtain a copy of the information in a bitmap shape's geometry.

```
gxBitmap *GXGetBitmap(gxShape source, gxBitmap *data,
                     gxPoint *position);
```

<code>source</code>	A reference to the bitmap shape whose geometry you want to copy.
<code>data</code>	A pointer to a <code>gxBitmap</code> structure. On return, this structure contains information copied from the geometry of the bitmap shape.
<code>position</code>	A pointer to a <code>gxPoint</code> structure. On return, this structure indicates the position of the upper-left corner of the bitmap shape.

*function result* A pointer to a `gxBitmap` structure containing information from the the geometry of the bitmap shape. This value is the same as the value you provided in the `data` parameter.

### DESCRIPTION

The `GXGetBitmap` function copies the information from the geometry of the bitmap shape indicated by the `source` parameter to the `gxBitmap` structure pointed to by the `data` parameter and returns a pointer to this information as its function result. This function also copies the bitmap position from the bitmap geometry to the `gxPoint` structure pointed to by the `position` parameter.

You may specify `nil` for the `data` or `position` parameters. If you do, this function does return the corresponding information.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>illegal_type_for_shape</code>	(debugging version)

**SEE ALSO**

For information about the `gxBitmap` structure, see “The Bitmap Geometry Structure” beginning on page 5-62.

For information about pixel images, see “Bitmap Geometries” beginning on page 5-5.

To create a bitmap shape, use the `GXNewBitmap` function, which is described on page 5-66.

To change the geometry of a bitmap shape, use the `GXSetBitmap` function, which is described in the next section.

**GXSetBitmap**

---

You can use the `GXSetBitmap` function to change the information in the geometry of a bitmap shape.

```
void GXSetBitmap(gxShape target, const gxBitmap *data,
                 const gxPoint *position);
```

<code>target</code>	A reference to the bitmap shape whose geometry you want to change.
<code>data</code>	A pointer to a <code>gxBitmap</code> structure containing new information for the geometry of the target bitmap shape.
<code>position</code>	A pointer to a <code>gxPoint</code> structure indicating the new bitmap position for the target bitmap shape.

**DESCRIPTION**

The `GXSetBitmap` function uses information you provide both in the `gxBitmap` structure pointed to by the `data` parameter and the `gxPoint` structure pointed to by the `position` parameter to change the information in the geometry of the bitmap shape referenced by the `target` parameter. If the target shape is not a bitmap shape, this function converts the target shape to a bitmap shape before setting the geometry of the shape.

You can change only the bitmap position by creating a `gxPoint` structure, setting its fields to reflect the new position, passing a pointer to it in the `position` parameter, and setting the `data` parameter to `nil`.

You can change other information in the geometry of the target bitmap shape by providing new information in a `gxBitmap` structure and passing a pointer to this structure in the `data` parameter.

If the pixel image of the target bitmap shape was not allocated by QuickDraw GX (for example, if you allocated the pixel image yourself before calling the `GXNewBitmap` function), then the `GXSetBitmap` function simply replaces the information in the geometry of the target bitmap shape with information from the fields of the `gxBitmap` structure pointed to by the `data` parameter.

## Bitmap Shapes

However, if QuickDraw GX allocated the pixel image of the target bitmap shape, you can use this function to change the dimensions of the existing pixel image.

You can change the bitmap height by providing a new height in the `height` field of the `gxBitmap` structure. You can change the bitmap width by setting the `rowBytes` field to 0 and provide a new bitmap width in the `width` field of the bitmap geometry structure. In this case, QuickDraw GX calculates an appropriate number of bytes per row.

In either case, this function does not scale the original pixel image; instead, it changes the amount of memory allocated to hold the pixel image. If you decrease the dimensions of the pixel image, QuickDraw GX fits the pixels in the original pixel image into a smaller space in memory, thereby losing some of the original pixel values. If you increase the dimensions of the pixel image, QuickDraw GX allocates more memory (possibly moving the original pixel image), thereby adding uninitialized pixels to the pixel image.

If QuickDraw GX allocated the original pixel image, you can also change the pixel size of the bitmap shape. You provide the new pixel size in the `pixelSize` field of the `gxBitmap` structure and the `GXSetBitmap` function expands or compresses the image to fit in the new pixel size. If you specify a smaller pixel size than the original, this function redistributes the colors in the color space of the bitmap shape.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>size_of_bitmap_exceeds_implementation_limit</code>	
<code>parameter_is_nil</code>	(debugging version)
<code>invalid_pixelSize</code>	(debugging version)
<code>bitmap_height_negative</code>	(debugging version)
<code>bitmap_width_negative</code>	(debugging version)
<code>bitmap_height_negative</code>	(debugging version)
<code>bitmap_rowBytes_negative</code>	(debugging version)
<code>bitmap_rowBytes_too_small</code>	(debugging version)
<code>bitmap_rowBytes_not_aligned</code>	(debugging version)
<code>bitmap_ptr_not_aligned</code>	(debugging version)
<code>bitmap_rowBytes_must_be_specified_for_user_image_buffer</code>	(debugging version)
<code>colorSpace_out_of_range</code>	(debugging version)

**Warnings**

<code>shape_access_not_allowed</code>	(debugging version)
---------------------------------------	---------------------

## SEE ALSO

For information about the `gxBitmap` structure, see “The Bitmap Geometry Structure” beginning on page 5-62.

For information about pixel images, bitmap height, bitmap width, pixel size, and number of rows per byte, see “Bitmap Geometries” beginning on page 5-5.

## Bitmap Shapes

For a complete discussion of the QuickDraw GX color architecture, see the chapter “Color and Color-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.

To create a bitmap shape, use the `GXNewBitmap` function, which is described on page 5-66.

To obtain a copy of the information from the geometry of a bitmap shape, use the `GXGetBitmap` function, which is described on page 5-68.

## Editing Bitmaps

---

This section describes the functions you can use to examine and change information in the pixel image of a bitmap shape.

The `GXGetShapePixel` function allows you to examine the value of a single pixel. The `GXSetShapePixel` function allows you to change the value of a single pixel.

The `GXGetBitmapParts` function allows you to extract a rectangular section of one bitmap shape and encapsulate it in another bitmap shape. The `GXSetBitmapParts` function allows you to replace a rectangular section of one bitmap shape with the pixel image of another bitmap shape.

## GXGetShapePixel

---

You can use the `GXGetShapePixel` function to determine the pixel value and the pixel offset of a specific pixel in a bitmap shape.

```
long GXGetShapePixel(gxShape source, long x, long y,
                    gxColor *data, long *index);
```

<code>source</code>	A reference to the bitmap shape containing the pixel to examine.
<code>x</code>	The index of the column in which the pixel lies.
<code>y</code>	The index of the row in which the pixel lies.
<code>data</code>	A pointer to a <code>gxColor</code> structure. On return, this structure contains the color value of the specified pixel.
<code>index</code>	A pointer to a <code>long</code> value. On return, this value contains the color value of the specified pixel (if the pixel size of the bitmap is 16 or 32) or the specified pixel's index into the bitmap's color set (if the pixel size of the bitmap is 1, 2, 4, or 8).
<i>function result</i>	The index of the byte containing the specified pixel in the source bitmap's pixel image.

## Bitmap Shapes

## DESCRIPTION

The `GXGetShapePixel` function copies the pixel value of the pixel determined by the `x` and `y` parameters from the source bitmap shape into the `gxColor` structure pointed to by the `data` parameter.

If the source bitmap shape has the `gxKeepShapeDirect` shape attribute set, this function also determines the pixel offset of the specified pixel and returns it in the `long` value pointed to by the `index` parameter. This function also returns a pointer to this value as the function result.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

`out_of_memory`  
`shape_is_nil`

**Warnings**

`shape_does_not_contain_a_bitmap` (debugging only)

## SEE ALSO

For information about the `gxBitmap` structure, see “The Bitmap Geometry Structure” beginning on page 5-62.

For information about pixels, pixel values, and pixel offsets, see “Bitmap Geometries” beginning on page 5-5.

To examine more than a single pixel of a bitmap, use the `GXGetBitmapParts` function, which is described on page 5-74.

To change the value of a pixel, use the `GXSetShapePixel` function, which is described in the next section.

## **GXSetShapePixel**

---

You can use the `GXSetShapePixel` function to change the pixel value of a specific pixel in a bitmap shape.

```
void GXSetShapePixel(gxShape target, long x, long y,
                    const gxColor *newColor, long newIndex);
```

<code>target</code>	A reference to the bitmap shape containing the pixel to change.
<code>x</code>	The index of the column in which the pixel lies.
<code>y</code>	The index of the row in which the pixel lies.



## Bitmap Shapes

<code>newColor</code>	A pointer to a <code>gxColor</code> structure indicating the new pixel value of the specified pixel. You may specify <code>nil</code> for this parameter if the target bitmap shape has the <code>gxIndexedSpace</code> color space.
<code>newIndex</code>	An index into a color set. You may use this parameter to set the pixel value if the target bitmap shape has the <code>gxIndexedSpace</code> color space.

## DESCRIPTION

The `GXSetShapePixel` function sets the pixel value of a specific pixel in the target bitmap. The pixel is determined by the values you provide in the `x` and `y` parameters. The new pixel value is determined by the `newColor` or `newIndex` parameter:

- <sup>n</sup> If you provide a color value in the `newColor` parameter, this function sets the pixel value of the specified pixel to be the closest color available in the color space of the target bitmap shape—even if the target bitmap shape has the `gxIndexedSpace` color space.
- <sup>n</sup> Alternatively, and only if the target bitmap shape has the `gxIndexedSpace` color space, you may provide `nil` for the `newColor` parameter and provide in the `newIndex` parameter a new index into the color set of the bitmap shape.

This function posts a `functionality_unimplemented` error for disk-based bitmaps.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>point_does_not_intersect_bitmap</code>	(debugging only)
<code>functionality_unimplemented</code>	(debugging only)

**Warnings**

<code>shape_does_not_contain_a_bitmap</code>	(debugging only)
--	------------------

## SEE ALSO

For information about the `gxBitmap` structure, see “The Bitmap Geometry Structure” beginning on page 5-62.

For information about pixels, pixel values, and pixel offsets, see “Bitmap Geometries” beginning on page 5-5.

To change more than a single pixel of a bitmap, use the `GXSetBitmapParts` function, which is described on page 5-75.

To examine the value of a pixel, use the `GXGetShapePixel` function, which is described on page 5-71.

## GXGetBitmapParts

---

You can use the `GXGetBitmapParts` function to extract a rectangular section of pixels from a bitmap.

```
gxShape GXGetBitmapParts(gxShape source,
                        const gxLongRectangle *bounds);
```

**source**      A reference to the bitmap shape containing the pixels to extract.  
**bounds**      A pointer to a `gxRectangle` indicating which part of the bitmap to extract.

*function result*   A reference to a new bitmap shape containing only the extracted section of the source bitmap shape.

### DESCRIPTION

The `GXGetBitmapParts` function extracts the pixels whose row number and column number fall within the boundaries of the rectangle pointed to by the `bounds` parameter, encapsulates the extracted pixel image in a new bitmap shape, and returns a reference to the new bitmap shape as the function result.

The returned bitmap shape has the same pixel size as the source bitmap shape. The returned bitmap shape also shares the same color space, color set, and color profile as the source bitmap shape.

The pixel image of the returned bitmap is allocated in QuickDraw GX memory.

### ERRORS, WARNINGS, AND NOTICES

#### Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_is_nil</code>	
<code>parameter_out_of_range</code>	(debugging only)
<code>point_does_not_intersect_bitmap</code>	(debugging only)

#### Warnings

<code>shape_does_not_contain_a_bitmap</code>	(debugging only)
--	------------------

**SEE ALSO**

For examples using this function, see “Editing Part of a Bitmap” beginning on page 5-53.

For information about the `gxBitmap` structure, see “The Bitmap Geometry Structure” beginning on page 5-62.

For information about the `gxLongRectangle` structure, see “The Long Rectangle Structure” on page 5-64.

For information about pixels and pixel images, see “Bitmap Geometries” beginning on page 5-5.

To examine a single pixel of a bitmap, use the `GXGetShapePixel` function, which is described on page 5-71.

To change a section of a bitmap, use the `GXSetBitmapParts` function, which is described in the next section.

## **GXSetBitmapParts**

---

You can use the `GXSetBitmapParts` function to replace the pixel values in a rectangular subsection of a bitmap’s pixel image.

```
void GXSetBitmapParts(gxShape target, const gxRectangle *bounds,
                     gxShape bitmapShape);
```

`target`            A reference to the bitmap shape containing the pixels to replace.

`bounds`           A pointer to a `gxRectangle` structure indicating which part of the target bitmap to replace.

`bitmapShape`     A reference to a bitmap shape containing the pixel values to use when replacing the specified pixels in the target bitmap shape.

**DESCRIPTION**

The `GXSetBitmapParts` function copies the pixel values (starting at the upper-left corner of the pixel image) of the source bitmap shape (which is indicated by the `bitmapShape` parameter) to the pixel image of the target bitmap shape. The `bounds` parameter determines how many rows and columns this function copies and where in the target bitmap the function places the copied pixel values.

The pixel image of the source bitmap may not be smaller than the size indicated by the `bounds` parameter; that is, the number of rows and columns in the pixel image of the source bitmap shape may not be less than the height and width of the specified rectangle, respectively.

## Bitmap Shapes

The source and target bitmap shapes must have the same pixel size, color space, and color set.

This function posts a `functionality_unimplemented` error for disk-based bitmaps.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_is_nil</code>	
<code>parameter_out_of_range</code>	(debugging only)
<code>point_does_not_intersect_bitmap</code>	(debugging only)
<code>functionality_unimplemented</code>	(debugging only)

**Warnings**

<code>shape_does_not_contain_a_bitmap</code>	(debugging only)
--	------------------

## SEE ALSO

For examples using this function, see “Editing Part of a Bitmap” beginning on page 5-53.

For information about the `gxBitmap` structure, see “The Bitmap Geometry Structure” beginning on page 5-62.

For information about the `gxLongRectangle` structure, see “The Long Rectangle Structure” on page 5-64.

For information about pixels and pixel images, see “Bitmap Geometries” beginning on page 5-5.

To change the pixel value of a single pixel, use the `GXSetShapePixel` function, which is described on page 5-72.

To extract a rectangular subsection of a bitmap, use the `GXGetBitmapParts` function, which is described on page 5-74.

## Drawing Bitmaps

---

QuickDraw GX provides two methods of drawing a bitmap:

- n You can create a bitmap shape (by calling the `GXNewBitmap` function, by copying an existing bitmap shape, and so on) and use the `GXDrawShape` function to draw the bitmap.
- n You can create a `gxBitmap` structure and use the `GXDrawBitmap` function to draw the bitmap.

## Bitmap Shapes

In general, you should use the `GXDrawShape` function to draw any QuickDraw GX graphic, including bitmap shapes. In fact, the `GXDrawBitmap` function creates a temporary bitmap shape, uses the `GXDrawShape` function to draw it, and then disposes of it. The `GXDrawShape` function is described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

You would typically use the `GXDrawBitmap` function only in simple situations—for example, if you knew you wanted to draw a particular bitmap only once.

## GXDrawBitmap

---

You can use the `GXDrawBitmap` function to draw a bitmap without encapsulating the bitmap geometry in a bitmap shape.

```
void GXDrawBitmap(const gxBitmap *data, const gxPoint *position);
```

<code>data</code>	A pointer to a <code>gxBitmap</code> structure that specifies information about the bitmap you want to draw.
<code>position</code>	A pointer to a <code>gxPoint</code> structure which indicates the position to draw the bitmap.

### DESCRIPTION

The `GXDrawBitmap` function allows you to draw a bitmap without having to create a bitmap shape yourself. Instead, you create a `gxBitmap` structure specifying the bitmap you want to draw and a `gxPoint` structure indicating the position of the bitmap, and then you pass a pointer to these structures in the `data` and `position` parameters, respectively.

The `GXDrawBitmap` function calls the `GXNewBitmap` function to create a temporary bitmap shape using the values specified in these structures and the style, ink and transform of the default bitmap shape. Then the `GXDrawBitmap` function draws the bitmap shape using the `GXDrawShape` function.

For information about how QuickDraw GX creates bitmap shapes using the values you provide in the fields of the `gxBitmap` structure, see the description of the `GXNewBitmap` function on page 5-66.

## Bitmap Shapes

## ERRORS, WARNINGS, AND NOTICES

**Errors**

out_of_memory	
shape_is_nil	
parameter_is_nil	
size_of_bitmap_exceeds_implementation_limit	
parameter_is_nil	(debugging version)
invalid_pixelSize	(debugging version)
bitmap_height_negative	(debugging version)
bitmap_width_negative	(debugging version)
bitmap_height_negative	(debugging version)
bitmap_rowBytes_negative	(debugging version)
bitmap_rowBytes_too_small	(debugging version)
bitmap_rowBytes_not_aligned	(debugging version)
bitmap_ptr_not_aligned	(debugging version)
bitmap_rowBytes_must_be_specified_for_user_image_buffer	(debugging version)
colorSpace_out_of_range	(debugging version)

**Warnings**

shape_access_not_allowed	(debugging version)
--------------------------	---------------------

## SEE ALSO

For examples of this function, see “Creating Black-and-White Bitmaps” beginning on page 5-15.

For information about the `gxBitmap` structure, see “The Bitmap Geometry Structure” beginning on page 5-62.

To encapsulate a bitmap geometry in a bitmap shape, use the `GXNewBitmap` function, which is described on page 5-66.

To draw a bitmap once you’ve encapsulated it in a bitmap shape, use the `GXDrawShape` function, which is described in the “Shape Objects” chapter of *Inside Macintosh: QuickDraw GX Objects*.

## Checking Bitmap Colors

---

QuickDraw GX provides the `GXCheckBitmapColor` function to allow you to determine which pixels in a bitmap are in the gamut of a specified color space or exactly match a color in a color set.

### `GXCheckBitmapColor`

---

You can use the `GXCheckBitmapColor` function to determine whether the color values in a bitmap's pixel image are in the gamut of a given color space or exactly match colors in a given color set.

```
gxShape GXCheckBitmapColor(gxShape source,
                           const gxLongRectangle *area,
                           gxColorSpace space, gxColorSet aSet,
                           gxColorProfile profile);
```

<code>source</code>	A reference to the bitmap shape whose pixels you want to check.
<code>area</code>	A pointer to a long rectangle specifying the area of the bitmap to check. You can specify a value of <code>nil</code> for this parameter to check the entire bitmap.
<code>space</code>	The color space to check the pixel values of the source bitmap against. You can specify the <code>gxIndexedSpace</code> color space to indicate that you want to test the pixel values against a color set.
<code>aSet</code>	A reference to the color set to check the pixel values of the source bitmap against.
<code>profile</code>	A pointer to the color profile to use when checking the pixel values of the source bitmap.
<i>function result</i>	A new bitmap shape with a pixel size of 1 bit per pixel. The value of each pixel in this bitmap indicates whether the color of the corresponding pixel in the source bitmap lies in the gamut of the specified color space (or, if you specified a color set, whether the color of the corresponding pixel exactly matches a color in that color set). If the corresponding source pixel does lie in the gamut of the color space (or match a color in the color set), the pixel value of this bitmap is set to 0, otherwise it is 1.

**DESCRIPTION**

The `GXCheckBitmapColor` function performs one of two tests on the pixels of the source bitmap:

- n If you specify a indexed color space in the `space` parameter, the function determines whether the color of the pixel exactly matches any color within the color set you provide in the `aSet` parameter.
- n If you specify any other color space in the `space` parameter, the function converts the pixel color to the indicated color space, using the color profile in the `profile` parameter, to determine whether the color is in the gamut represented by the color space and color profile.

If you specify `nil` as the `area` parameter, this function tests every pixel in the source bitmap's pixel image. If you provide a pointer to a long rectangle in this parameter, the function only tests the pixels that fall within the corresponding rectangular subsection of the source bitmap.

This function returns as the function result a 1 bit-per-pixel bitmap shape with a bitmap height and bitmap width corresponding to the dimensions of the `area` parameter. Each pixel in the returned bitmap is set to a value of 0 if the corresponding pixel in the source bitmap passed the test. The pixel value is 1 otherwise.

**ERRORS, WARNINGS, AND NOTICES****Errors**

`out_of_memory`  
`color_is_nil`  
`colorSpace_out_of_range` (debugging version)

**Warnings**

`colorSet_index_out_of_range` (debugging version)

**SEE ALSO**

For information about the `gxLongRectangle` structure, see page 5-64.

For information about colors, color spaces, color sets, color profiles, and the `GXCheckColor` function, see chapter “Colors and Color-Related Objects” in *Inside Macintosh: QuickDraw GX Objects*.



## Summary of Bitmap Shapes

---

### Constants and Data Types

---

#### The Bitmap Geometry Structure

```
typedef struct {
    char        *image;    /* pointer to the pixel image */
    long         width;     /* bitmap width */
    long         height;    /* bitmap height */
    long         rowBytes;  /* number of bytes per row */
    long         pixelSize; /* number of bits per pixel */
    gxColorSpace space;     /* color space used to interpret pixel values */
    gxColorSet   set;       /* color set to use to interpret pixel values */
    gxColorProfile profile; /* color matching information */
} gxBitmap;
```

#### The Long Rectangle Structure

```
struct gxLongRectangle {
    long    left;
    long    top;
    long    right;
    long    bottom;
};
```

#### Constants For Bitmaps With Disk-Based Pixel Images

```
#define gxBitmapFileAliasImageValue    0x00000001

#define gxBitmapFileAliasTagType       'bfil'
```

#### Bitmap Data Source Alias Structure

```
struct gxBitmapDataSourceAlias {
    unsigned long fileOffset;          /* file offset (in bytes) */
    unsigned long aliasRecordSize;     /* size of alias record */
    unsigned char aliasRecord[gxAnyNumber]; /* alias record */
};
```

## Functions

---

### Creating Bitmaps

```
gxShape GXNewBitmap          (const gxBitmap *data, const gxPoint *position);
```

### Getting and Setting Bitmap Geometries

```
gxBitmap *GXGetBitmap        (gxShape source, const gxBitmap *data,  
                               const gxPoint *position);
```

```
void GXSetBitmap              (gxShape target, const gxBitmap *data,  
                               const gxPoint *position);
```

### Editing Bitmaps

```
long GXGetShapePixel          (gxShape source, long x, long y, gxColor *data,  
                               long *index);
```

```
void GXSetShapePixel          (gxShape target, long x, long y,  
                               const gxColor *newColor, long newIndex);
```

```
gxShape GXGetBitmapParts      (gxShape source, const gxLongRectangle *bounds);
```

```
void GXSetBitmapParts         (gxShape target, const gxLongRectangle *bounds,  
                               gxShape bitmapShape);
```

### Drawing Bitmaps

```
void GXDrawBitmap             (const gxBitmap *data, const gxPoint *position);
```

### Checking Bitmap Colors

```
gxShape GXCheckBitmapColor    (gxShape source,  
                               const gxLongRectangle *area,  
                               gxColorSpace space, gxColorSet aSet,  
                               gxColorProfile profile);
```

# Picture Shapes

---

## Contents

About Picture Shapes	6-3
Overriding Styles, Inks, and Transforms	6-8
Multiple References	6-10
Unique Items Shape Attribute	6-15
Picture Hierarchies	6-18
Transform Concatenation	6-19
About Hit-Testing Picture Shapes	6-24
Using Picture Shapes	6-26
Creating and Drawing Picture Shapes	6-27
Getting and Setting Picture Geometries	6-31
Adding Items to a Picture	6-32
Removing and Replacing Items in a Picture	6-35
Using Overriding Styles, Inks, and Transforms	6-38
Adding Multiple References	6-40
Adding Items With the Unique Items Attribute Set	6-43
Creating Picture Hierarchies	6-44
Hit-Testing Pictures	6-46
Applying Functions Described Elsewhere to Picture Shapes	6-52
Functions That Post Errors or Warnings When Applied to Pictures	6-52
Shape-Related Functions Applicable to Pictures	6-54
Geometric Operations Applicable to Pictures	6-55
Style-Related Functions Applicable to Pictures	6-55
Ink-Related Functions Applicable to Pictures	6-56
Transform-Related Functions Applicable to Pictures	6-56
Picture Shapes Reference	6-57
Functions	6-57
Creating Picture Shapes	6-57
GXNewPicture	6-57

Getting and Setting Picture Geometries	6-59
GXGetPicture	6-59
GXSetPicture	6-61
Editing Picture Parts	6-63
GXGetPictureParts	6-63
GXSetPictureParts	6-65
Drawing Pictures	6-67
GXDrawPicture	6-67
Hit-Testing Pictures	6-69
GXHitTestPicture	6-69
Summary of Picture Shapes	6-72
Functions	6-72

## Picture Shapes

This chapter describes picture shapes and the functions you use to manipulate them. It also discusses the functions described in other chapters that you can apply to picture shapes.

In particular, this chapter shows you how you can create and draw picture shapes; edit a picture shape's list of items; override style, ink, and transform information for items in a picture; create picture hierarchies; and hit-test picture shapes.

You should be familiar with the information in the chapter "Shape Objects" of *Inside Macintosh: QuickDraw GX Objects* before you read this chapter, and you will probably want to be familiar with the information in the chapter "Transform Objects" of that book. You might also want to be familiar with the other shape types, which are described in Chapter 2, "Geometric Shapes," and Chapter 5, "Bitmap Shapes," of this book, as well as in the chapter "Typographic Shapes" of *Inside Macintosh: QuickDraw GX Typography*.

## About Picture Shapes

---

A **picture shape** represents a collection of other shapes. For example, you could create a scroll bar using a picture shape:

- n You could create separate polygon shapes to represent the scroll box, the gray area, and the two scroll arrows.
- n You could then collect these individual polygon shapes into a single picture shape to represent the entire scroll bar.

Using picture shapes, you can create complex graphics, create shapes with both graphic and typographic content, combine multiple bitmaps into a single shape, create groups of shapes, create shape layers, group shapes into pages to prepare for printing, and so on.

Like any QuickDraw GX shape, a picture shape is represented in memory by a shape object, a style object, an ink object, and a transform object. A shape object representing a picture shape contains the same properties as a shape object representing a geometric or a typographic shape: owner count, tag list, shape type, shape fill, geometry, and so on.

Since picture shapes contain other shapes, they don't make much use of their shape fill property, although you can specify a no-fill shape fill if you don't want the picture to appear when drawn.

Picture shapes also don't make much use of their associated style object, since each shape in the picture has its own style object.

Pictures shapes also don't make much use of their ink objects for the same reasons.

Picture shapes do make full use of their transform objects, however. For example, you can scale, skew, rotate, and clip picture shapes as a whole, as well as separately for each individual shape in the picture. This process is described in more detail in the section "Transform Concatenation" beginning on page 6-19.

Picture shapes differ from other types of shapes primarily in the content of their geometries. A picture shape's geometry contains a list of picture items. Each **picture item** contains a reference to another shape.

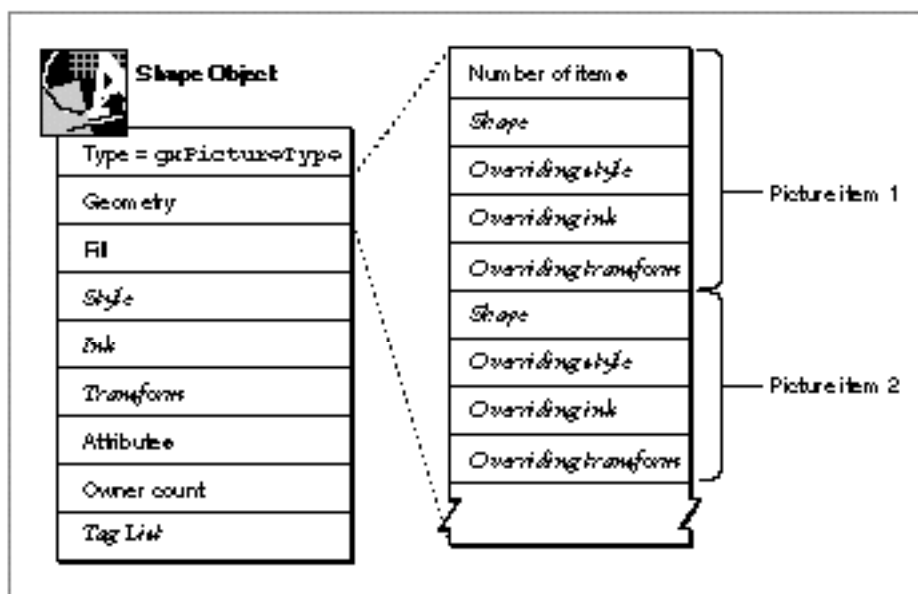
## Picture Shapes

Although each of the shapes in a picture has its own style, ink, and transform object, picture shapes allow you to provide an overriding style, ink, and transform object for each of these shapes. QuickDraw GX uses this overriding information only when drawing the picture. Even after you insert a shape into a picture, you can still draw the original shape using its original style, ink, and transform object.

Overriding objects are described in the next section “Overriding Styles, Inks, and Transforms” beginning on page 6-8.

Figure 6-1 shows a graphic representation of a picture shape and a picture geometry.

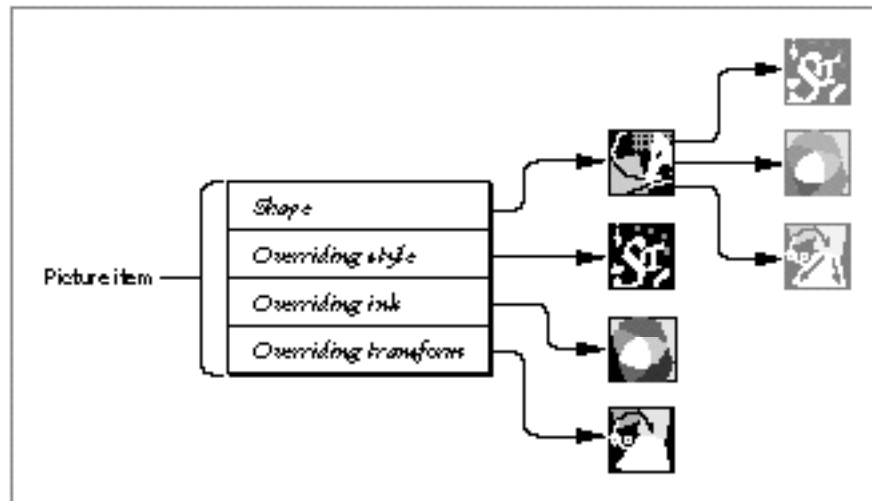
**Figure 6-1** A picture shape



## Picture Shapes

Figure 6-2 shows a single picture item. This item contains a reference to a shape object, which contains a reference to its associated style, ink, and transform objects. These objects are shown in grey, because the picture item also contains references to an overriding style, ink, and transform object for the shape.

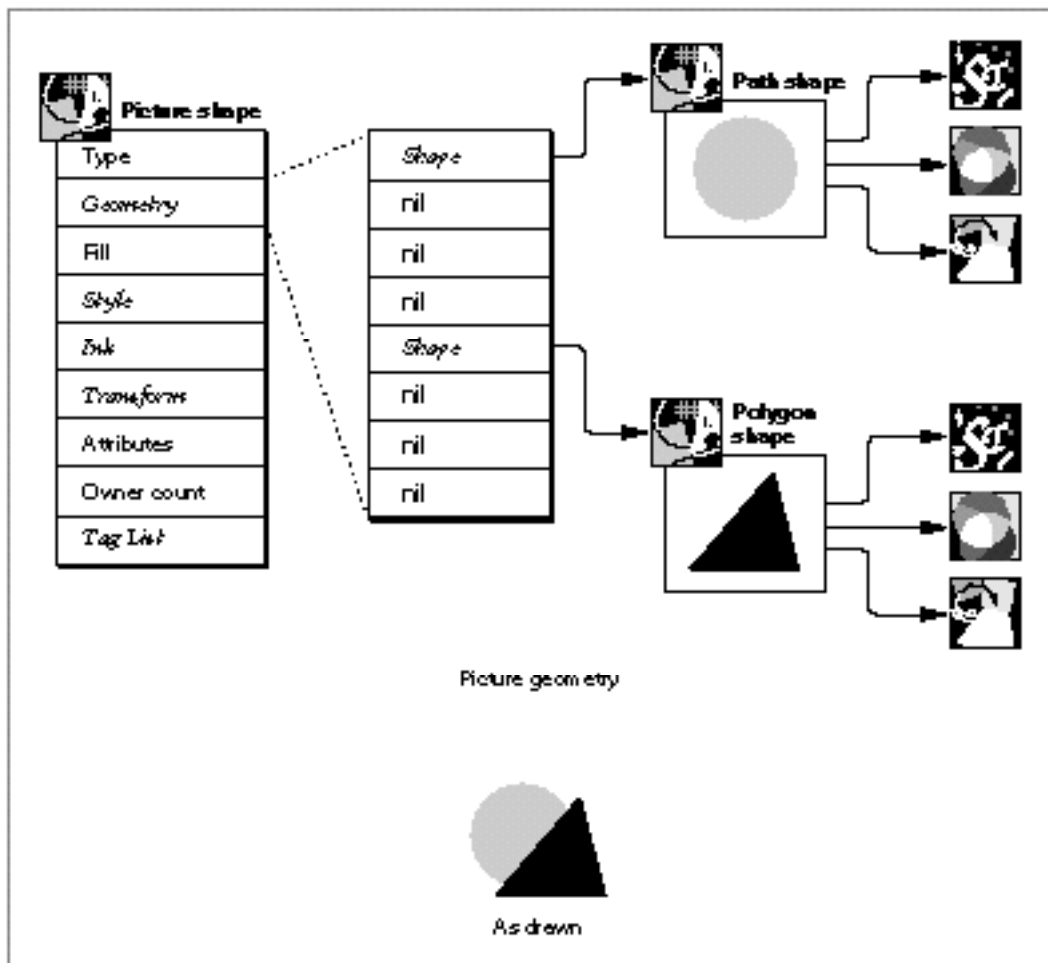
**Figure 6-2** A picture item



## Picture Shapes

Figure 6-3 shows an example of a picture shape with a geometry that contains two picture items. Each item contains a reference to a shape, but neither item contains a reference to an overriding style, ink, or transform object. Therefore, when QuickDraw GX draws this picture, it draws each shape in the picture using the style, ink, and transform information originally associated with the shape, as shown at the bottom of Figure 6-3.

**Figure 6-3** A picture geometry with two items



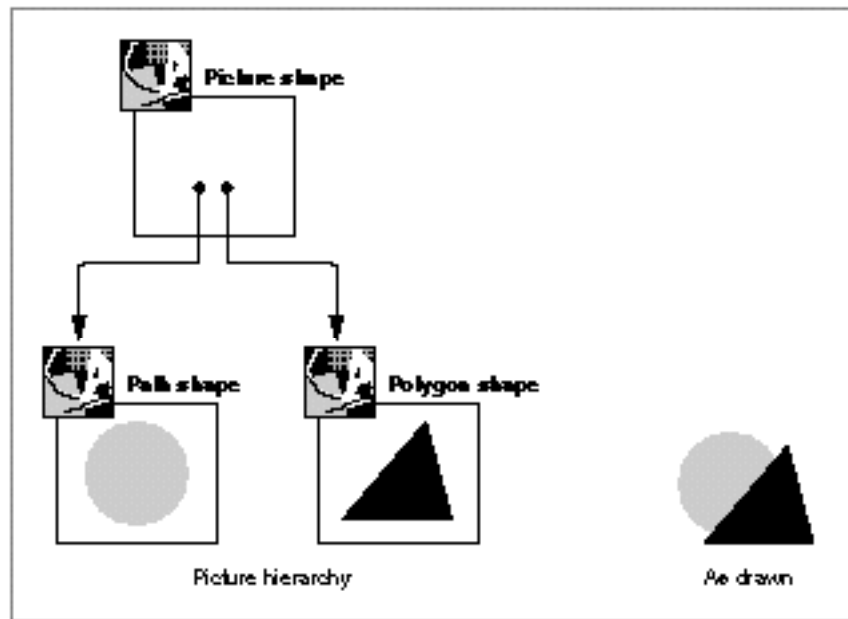


## Picture Shapes

Notice that QuickDraw GX draws the shapes in a picture in the order the references to them appear in the picture geometry: from back to front.

Figure 6-3 shows the entire shape object and picture geometry for the picture shape. Figure 6-4 shows a condensed view of the same picture. This chapter uses condensed views of picture shapes when drawing picture hierarchies, which are described in “Picture Hierarchies” beginning on page 6-18.

**Figure 6-4** Condensed view of picture with two items

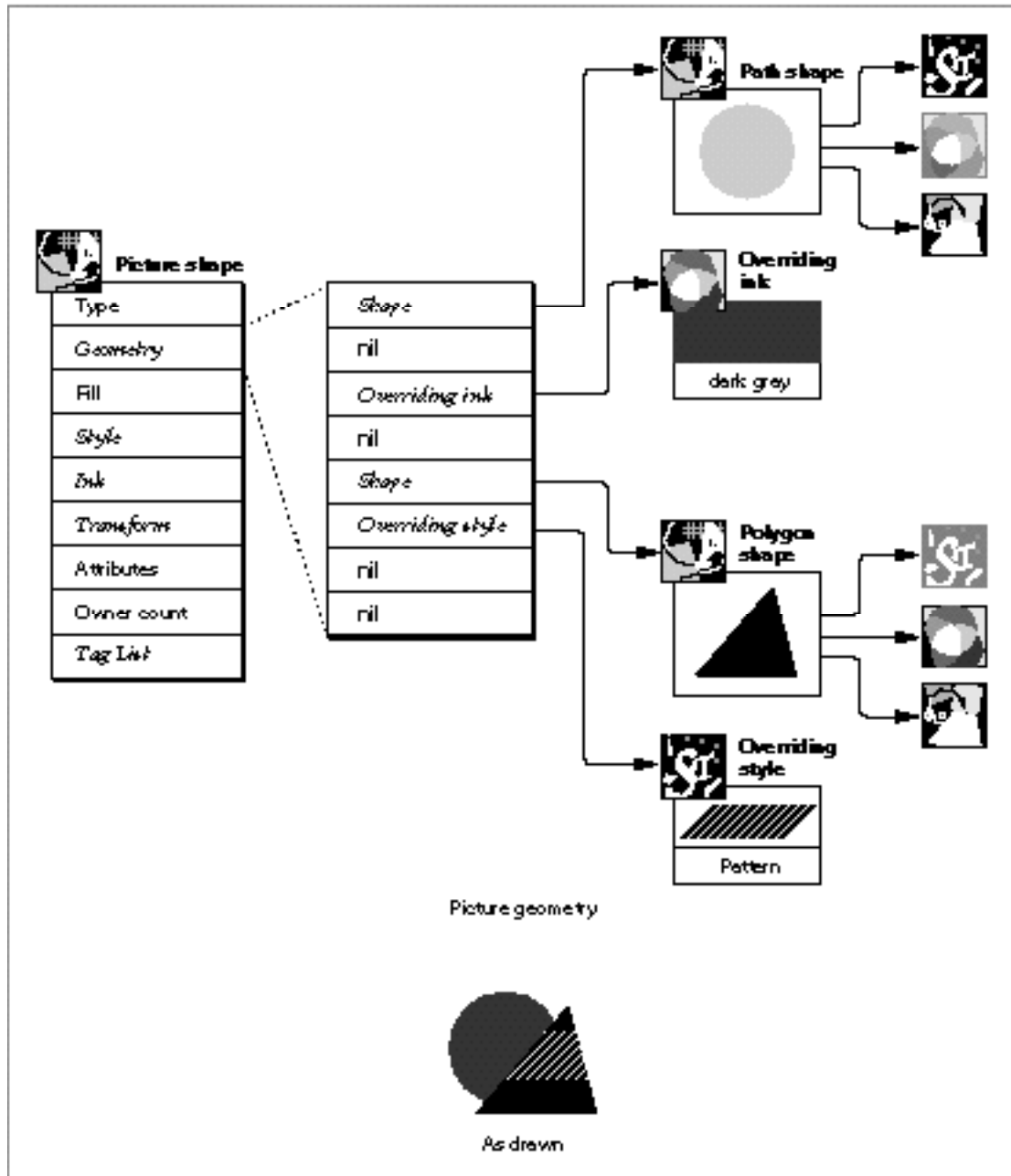


## Overriding Styles, Inks, and Transforms

---

QuickDraw GX allows you to specify an **overriding style, ink, or transform** object for any item in a picture. If an item has an overriding style, ink, or transform object, QuickDraw GX uses the information in the overriding object rather than the information in the original style, ink, or transform when drawing that item of the picture shape.

Figure 6-5 shows the picture from Figure 6-4 with overriding information added. In this figure, the first picture item has an overriding ink, which specifies a dark gray color. The second picture item has an overriding style, which specifies a pattern.

**Figure 6-5** A picture shape with overrides

When QuickDraw GX draws the picture represented in Figure 6-5, it draws the first picture item using the information in the overriding ink object, rather than the information in the ink object originally associated with the first item. Similarly, when it draws the second shape, it uses the information in the overriding style rather than the information in the original style.

## Multiple References

QuickDraw GX allows multiple items in a picture to reference the same shape. Figure 6-6 shows an example of a picture shape containing four items. In this example, each item references the same shape: a black rectangle.

**Figure 6-6** A picture containing multiple references to the same shape

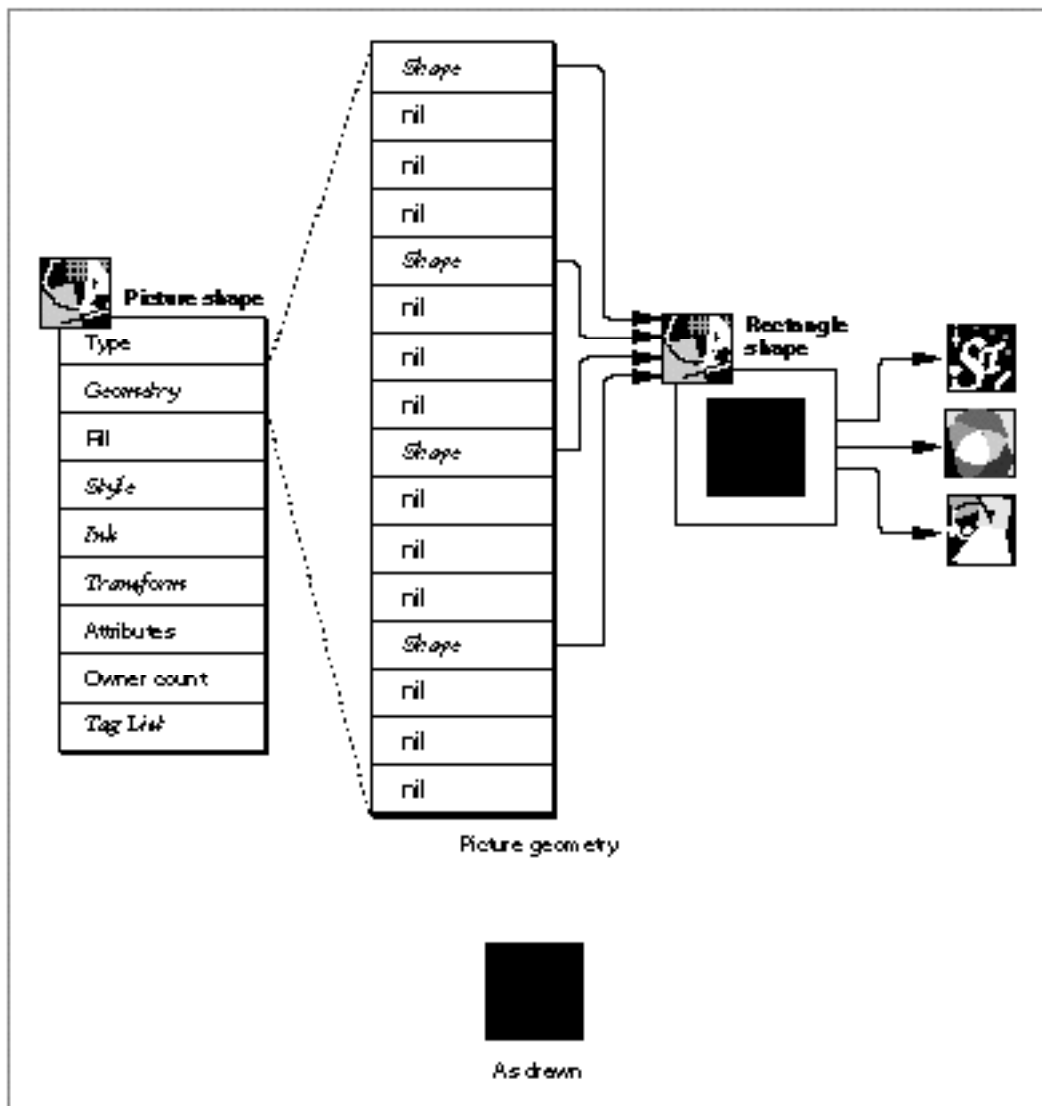
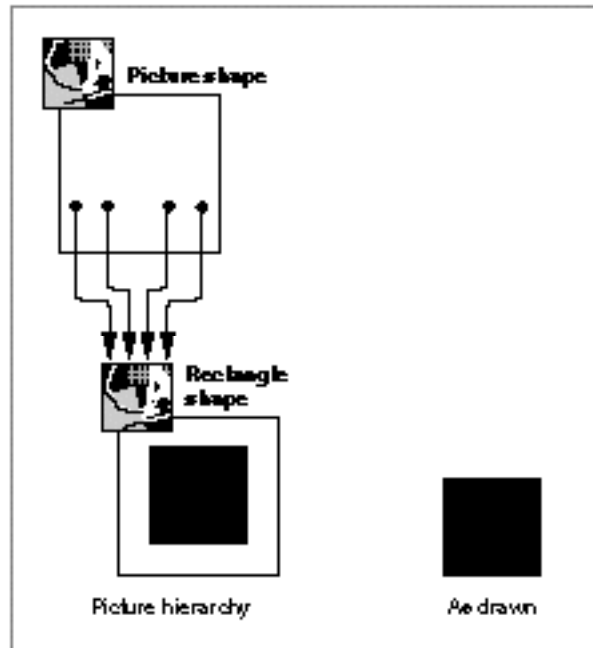


Figure 6-7 shows the condensed view of the picture from Figure 6-6.

**Figure 6-7** A condensed view of a picture with multiple references

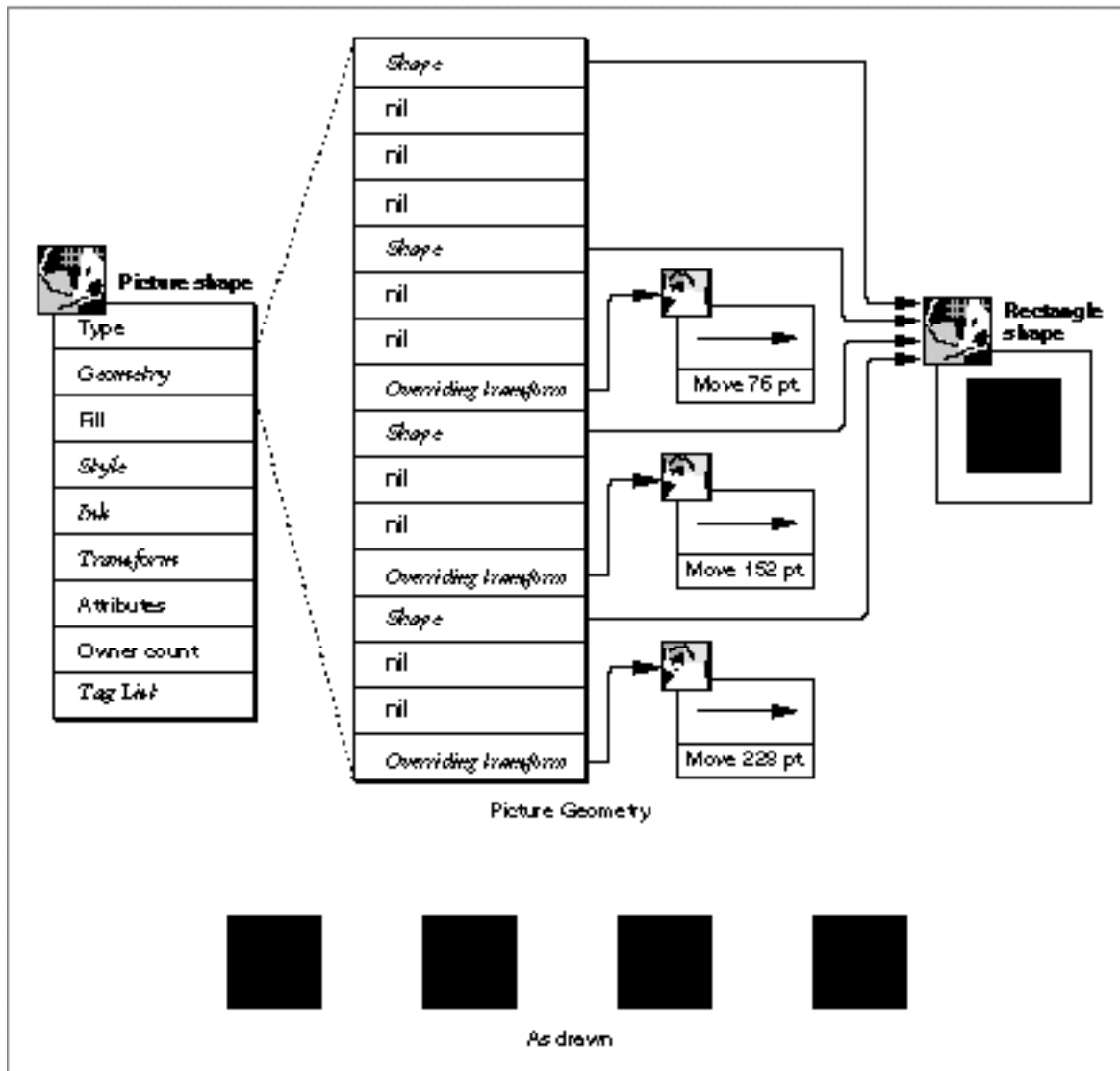


Although the picture shape shown in Figure 6-7 contains four references to the black rectangle, only one black rectangle appears when the picture is drawn. You might expect the rectangle to be drawn four times; however, it only appears once because the rectangle is redrawn in the same location four times.

## Picture Shapes

Having multiple references to the same shape becomes more useful when you add overriding information. For example, if you add overriding transforms to three of the items in the picture shape from Figure 6-6, all four items appear when the picture is drawn, as shown in Figure 6-8.

**Figure 6-8** Multiple references with overriding transforms



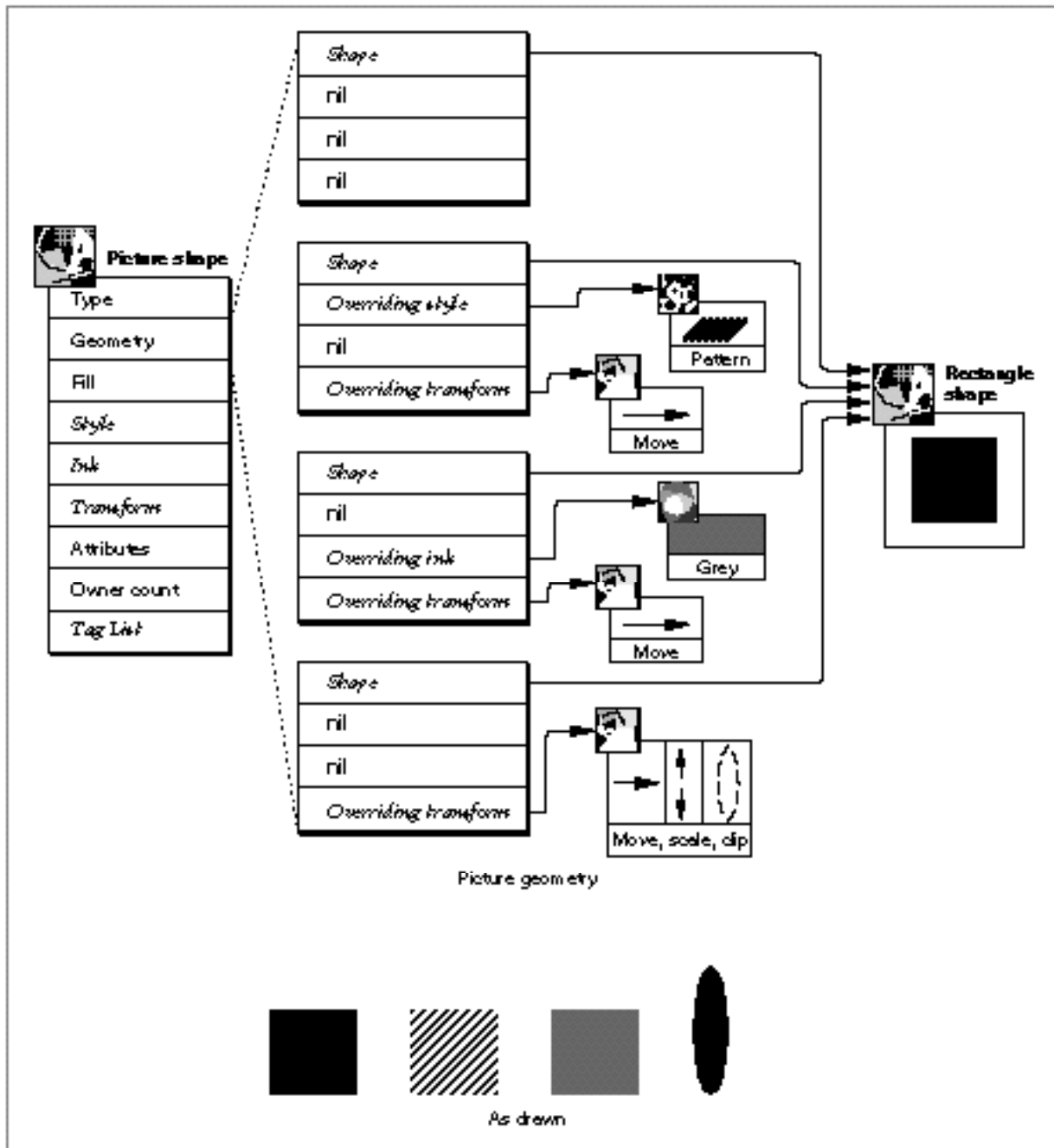
Picture Shapes

The picture shape in Figure 6-8 contains four items each referencing the same black rectangle shape. However, the second, third, and fourth items contain overriding transforms. When drawing this picture shape, QuickDraw GX applies the original transform when drawing the first item, and applies the overriding transforms when drawing the second, third, and fourth items. In this way, the four items appear separate when the picture is drawn, even though all four items reference the same shape.

## Picture Shapes

You can use overriding styles and inks to make multiple references even more powerful. In Figure 6-9, the second item has an overriding style as well as an overriding transform, the third item has an overriding ink as well as an overriding transform, and the fourth item has an overriding transform that not only moves, but scales and clips as well.

**Figure 6-9** Multiple references with overriding styles, inks, and transforms





## Picture Shapes

Although each item in the picture shape shown in Figure 6-9 references the same black rectangle, the use of overriding styles, inks, and transforms creates substantial variations in the items of the picture as drawn.

For more examples of multiple references and overriding styles, inks, and transforms, see “Adding Multiple References” beginning on page 6-40.

## Unique Items Shape Attribute

---

One of the shape attributes provided by QuickDraw GX is the **unique items attribute**. This attribute affects the way shapes are added to a picture:

- n If a picture shape does not have the unique items attribute set, QuickDraw GX adds shapes to the picture by reference.
- n If a picture shape does have the unique items attribute set, QuickDraw GX adds shapes to the picture by copying the shapes and adding a reference to the copy.

Although you may clear the unique items attribute for a picture at any time, you may set the unique items attribute only when a picture is empty—that is, only when the picture contains no items.

You set or clear the unique items attribute using the `GXGetShapeAttributes` function, which is described in the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects*.

Figure 6-10 depicts an empty picture shape and a polygon shape. The following two figures use these shapes to illustrate the effect of the unique items attribute.

---

**Figure 6-10** An empty picture shape and a polygon shape

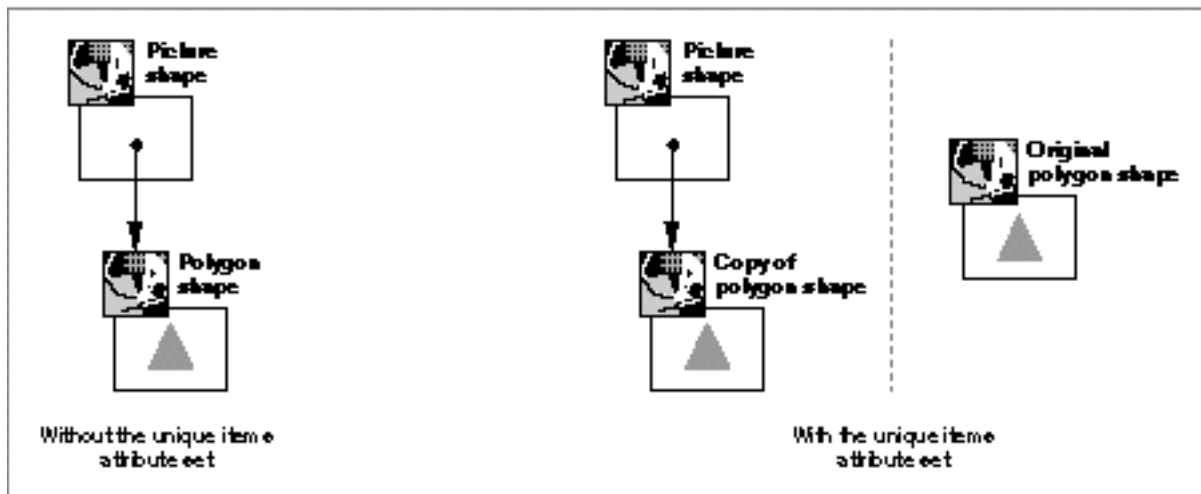


## Picture Shapes

Figure 6-11 shows the result of adding the polygon shape to the empty picture shape. In the left half of this figure, the picture shape does not have the unique items attribute set. In this case, adding the polygon shape to the empty picture simply adds a reference to the polygon shape to the geometry of the picture shape and increases the owner count of the polygon shape.

In the right half of this figure, the picture shape has the unique items attribute set. In this case, adding the polygon shape to the empty picture creates a deep copy of the polygon shape—including all objects referenced by the polygon shape—and adds the copy to the geometry of the picture shape. The original polygon shape is unchanged.

**Figure 6-11** Adding a polygon shape to a picture shape



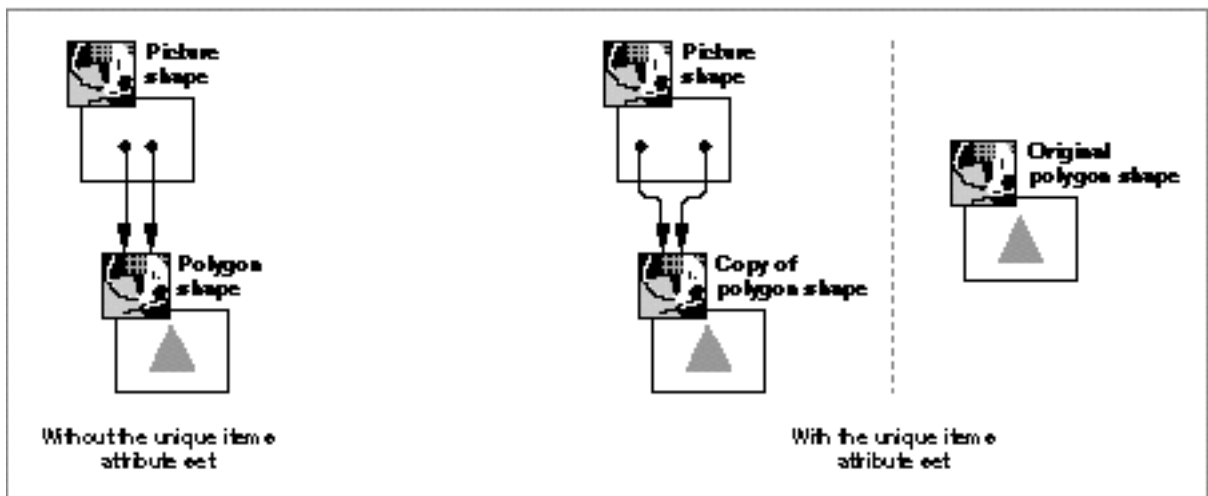
## Picture Shapes

Figure 6-12 shows the result of adding the same polygon shape to the empty picture shape twice.

In the left half of this figure, the picture shape does not have the unique items attribute set. In this case, the first time the polygon shape is added to the empty picture, a reference to the polygon shape is added to the geometry of the picture shape and the owner count of the polygon shape is incremented. The second time the polygon is added to the picture, another reference to the polygon is added to the picture geometry and the owner count of the polygon is incremented again.

In the right half of this figure, the picture shape has the unique items attribute set. In this case, the first time the polygon shape is added to the empty picture, QuickDraw GX creates a deep copy of the polygon shape—including all objects referenced by the polygon shape—and adds a reference to the copy to the geometry of the picture shape. The original polygon shape is unchanged. The second time the polygon is added to the picture, QuickDraw GX notices that the polygon has already been added to the picture and has not been changed. Therefore, to avoid making a second deep, QuickDraw GX simply adds to the picture geometry another reference to the first deep copy.

Figure 6-12 Adding a shape to a picture twice



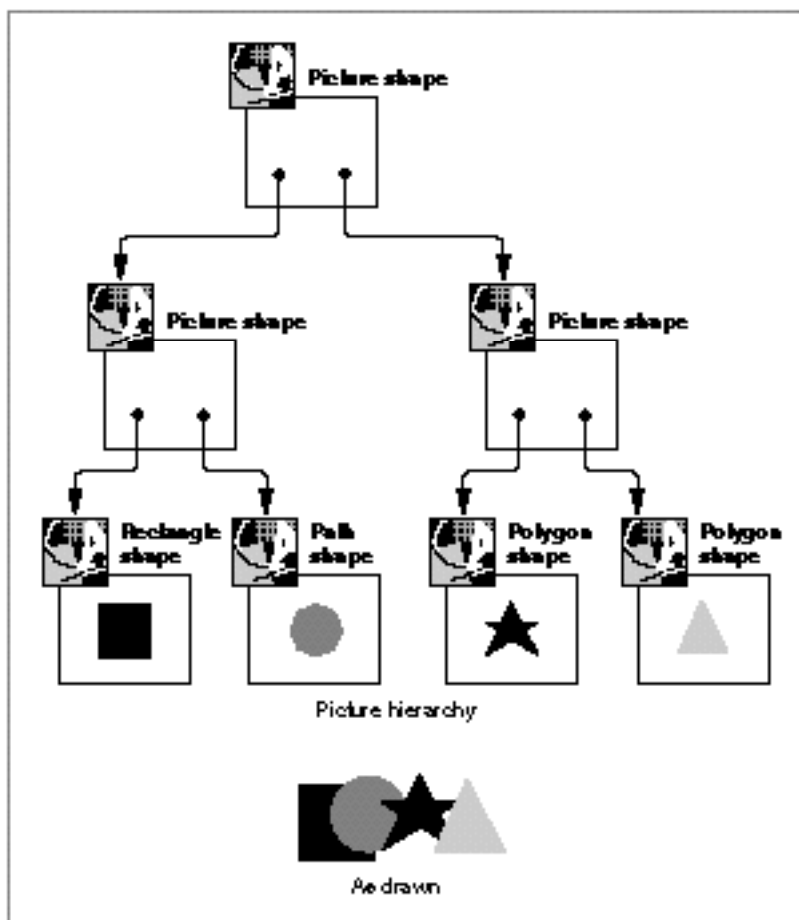
For more examples involving the unique items attribute, see “Adding Items With the Unique Items Attribute Set” beginning on page 6-43.

## Picture Hierarchies

Each item of a picture shape contains a reference to another shape. These shapes can be of any type, including other picture shapes. When a picture shape contains references to other picture shapes, you have a **picture hierarchy**. Figure 6-13 shows a picture hierarchy.

Figure 6-13 depicts a picture shape with two items. Each item references another picture shape, each of which also has two items. This figure shows the condensed view of the picture hierarchy.

**Figure 6-13** A condensed view of a picture hierarchy



Each item in a picture hierarchy has a **level**. The two items belonging to the topmost picture shape—which are picture shapes themselves—have a level of 1. Items belonging to pictures that have a level of 1 have a level of 2, and so on. In the picture hierarchy shown in Figure 6-13, the four geometric shapes all have a level of 2.

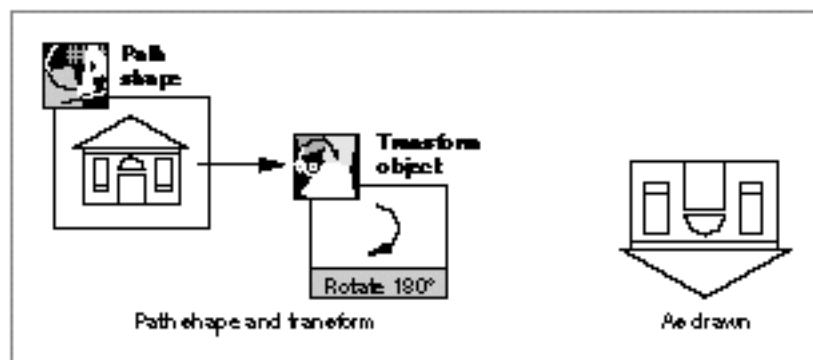
## Transform Concatenation

Each item in a picture shape has its own transform object and possibly an overriding transform as well. When QuickDraw GX draws a picture shape, it maps and clips each item according to the mapping and clipping information stored in that item's transform object (or the information in the item's overriding transform, if it has one). After applying mappings and clippings to the individual items of a picture, QuickDraw GX applies a mapping and clipping to the entire picture, as indicated by the transform object associated with the picture shape. In this way, each item in the picture can go through two transformations: an individual transformation as indicated by the item's individual transform (or overriding transform), and a group transformation as indicated by the picture shape's transform. This process is called **transform concatenation**.

If a picture shape contains a picture hierarchy, QuickDraw GX repeats this concatenation process from the individual shapes at the lowest level of the hierarchy all the way up to the picture shape at the highest level of the hierarchy.

As an example, Figure 6-14 shows a path shape representing a house. This path shape has a transform that rotates it 180 degrees.

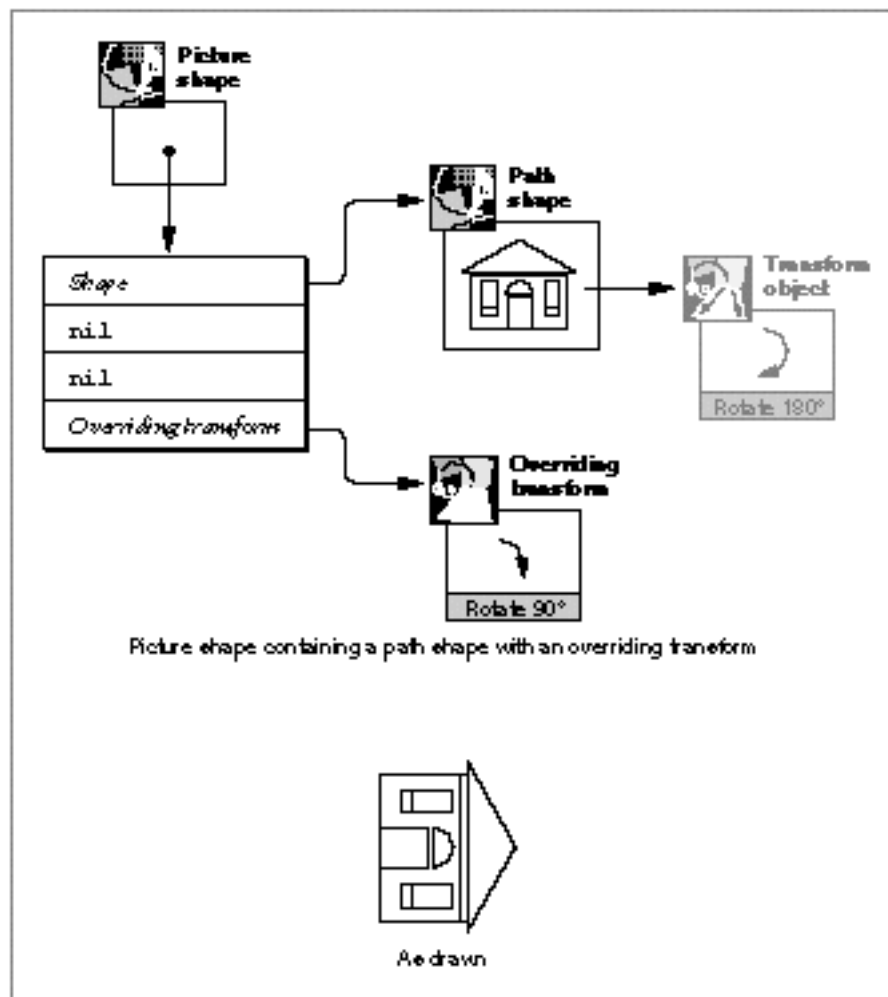
**Figure 6-14** A path shape and its transform



## Picture Shapes

Figure 6-15 shows the same path shape, but in this figure the path shape has been added to a picture shape as the picture's only item. This item includes an overriding transform. When drawing this picture, QuickDraw GX ignores the original transform, and rotates every item in the path shape clockwise by 90 degrees, as specified in the overriding transform.

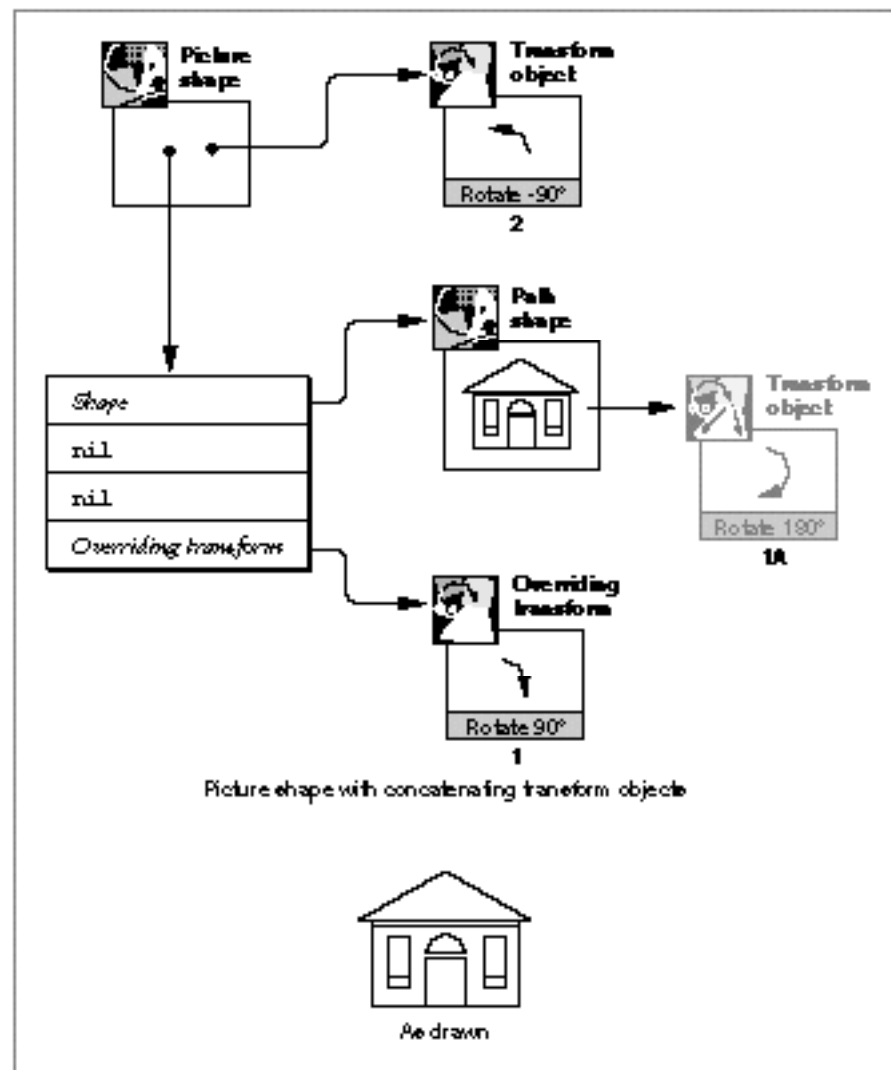
**Figure 6-15** A picture with an overriding transform



## Picture Shapes

Figure 6-16 shows the same picture shape as Figure 6-15. In Figure 6-16, however, the picture shape at the top of the picture hierarchy has its own transform object that specifies that the entire picture should be rotated counterclockwise by 90 degrees. QuickDraw GX concatenates the overriding transform of the path shape (labeled 1 in the picture) with the transform of the top of the picture hierarchy (labeled 2 in the picture), and draws the house at its original orientation. The original transform of the path shape (labeled 1A) is ignored because of the overriding transform.

**Figure 6-16** Simple transform concatenation



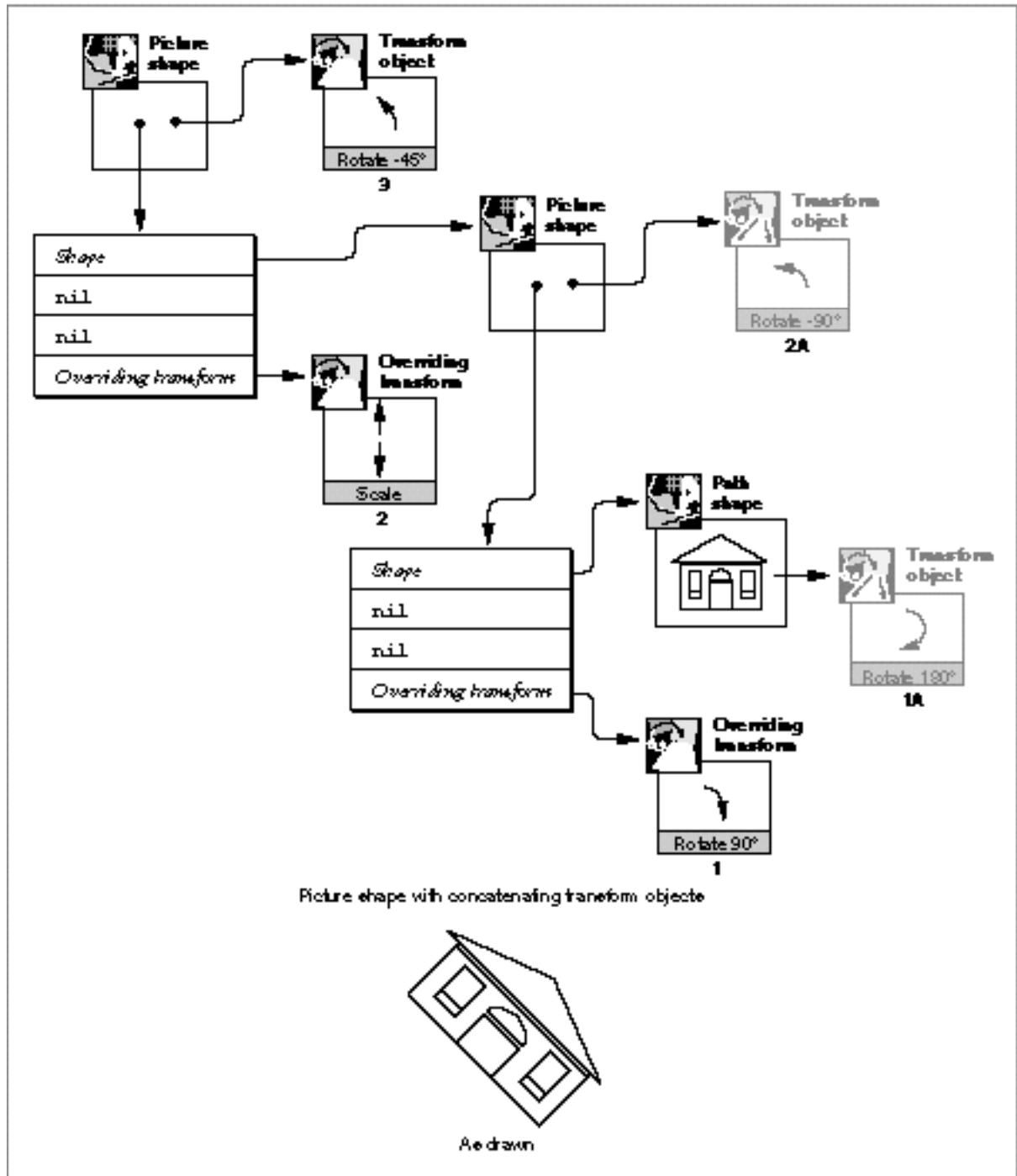
## Picture Shapes

Figure 6-17 shows an even more complex example of transform concatenation. This figure shows the same picture from Figure 6-16, but in Figure 6-17 this picture has been added as an item to another picture.

To draw this picture, QuickDraw GX uses the overriding transform (labeled 1) of the original path shape, which rotates it 90 degrees to the right. Then QuickDraw GX uses the overriding transform (labeled 2) associated with the picture that contains the path shape, which scales the picture by a factor of 2. Finally, QuickDraw GX uses the transform object (labeled 3) of the picture at the top of the hierarchy, which rotates the picture 45 degrees to the right. The result is shown at the bottom of Figure 6-17.



Figure 6-17 Intricate transform concatenation



You can find more examples of transform concatenation in “Creating Picture Hierarchies” beginning on page 6-44.

## About Hit-Testing Picture Shapes

---

When the user clicks the mouse, your application receives the information from the Macintosh Toolbox about where the mouse click occurred. By sending this information to the `GXHitTestPicture` function, you can find out which item in a picture was hit. This process is called hit-testing a picture shape.

When hit-testing a picture shape, QuickDraw GX searches through the shapes contained in the picture until it finds the shape that was hit by the hit-test point. As QuickDraw GX searches through the shapes in the picture, it

- n hit-tests the shape, using the hit-test information in that shape's transform object (or overriding transform object, if the shape has one) to determine if the shape was hit or not
- n determines whether the hit shapes satisfy criteria that you specify

QuickDraw GX returns information about the first item that was hit and satisfies the criteria.

Since more than one shape in a picture can be hit during a single hit-test, you provide QuickDraw GX with extra selection criteria when hit-testing a picture. Specifically, you specify a depth and a level:

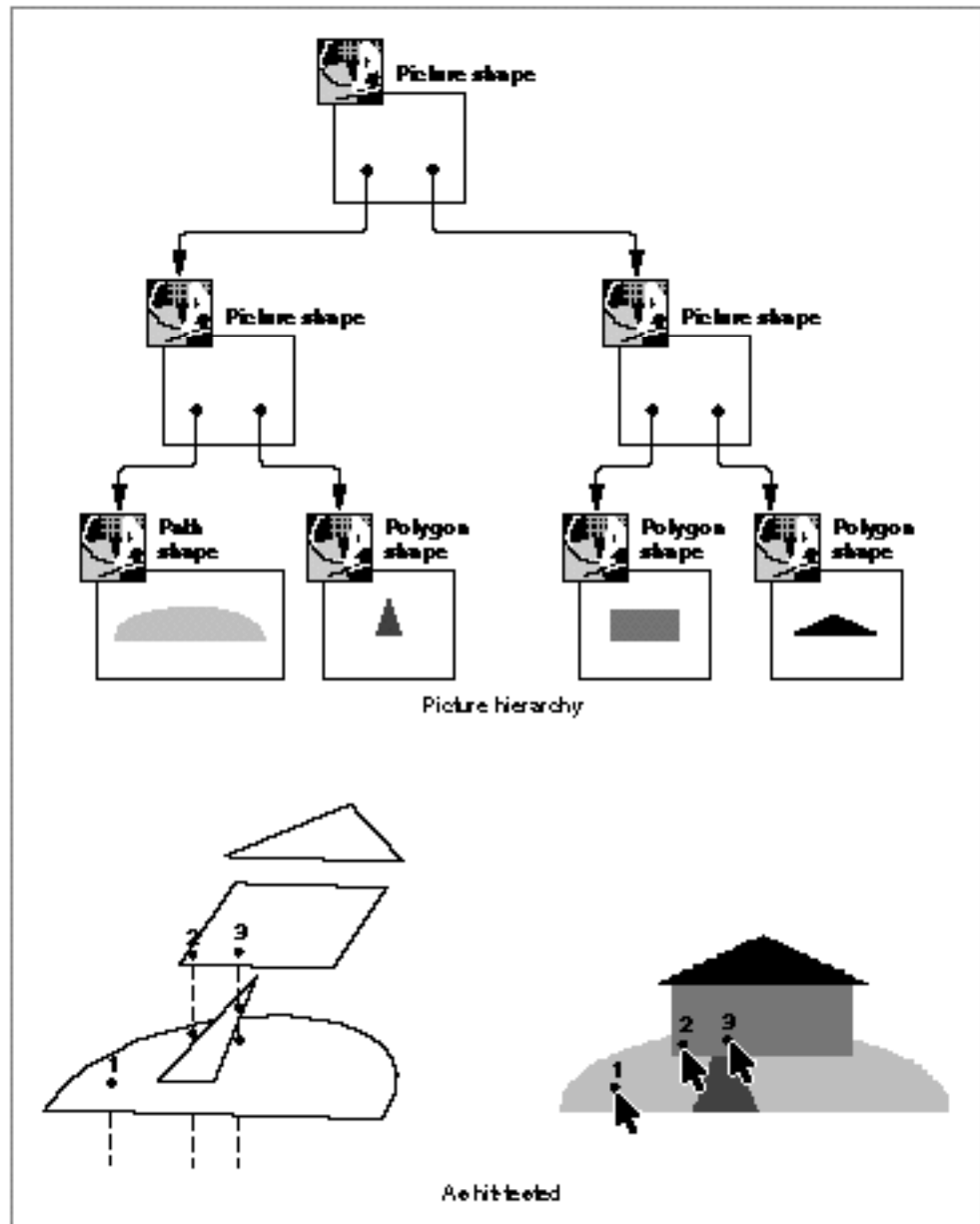
- n Pictures frequently contain shapes that overlap when drawn. Therefore, it is possible that the test point hits multiple shapes. For example, if the picture contains two shapes, one on top of the other, the test point might hit both of them. You can control which of these shapes QuickDraw GX selects as the hit shape by specifying a shape **depth**. In this example, specifying a shape depth of 1 would indicate that QuickDraw GX should select the shape that was drawn on top as the hit shape. Specifying a shape depth of 2 would indicate that QuickDraw GX should select the shape that was drawn underneath as the hit shape.
- n In a picture hierarchy, each shape can be contained by a picture shape, which in turn can be contained by another picture shape, and so on. If the hit shape has a level of 3, for example, you can specify that QuickDraw GX return a reference to the hit shape by specifying a level of 3. You can specify that QuickDraw GX return a reference to the picture that contains the hit shape by specifying a level of 2. You can specify that QuickDraw GX return a reference to the picture that contains the picture that contains the hit shape by specifying a level of 1.

## Picture Shapes

Figure 6-18 shows an example. The picture shape shown in this figure has two items, each of which are pictures. Each of these pictures has two items itself, making a total of four shapes that have a level of 2 in this hierarchy.

This figure shows the picture as drawn, and three sample hit-test points.

**Figure 6-18** A picture shape and hit-test points



## Picture Shapes

The first sample hit-test point hits only one shape: the lawn path shape. If you specified a depth of 1, QuickDraw GX would select this shape as the hit shape. The shape returned by QuickDraw GX, however, depends on what you specify for the level. If you specify 2, the lawn path shape would be returned. If you specified 1, however, the picture that contains the lawn path shape would be returned.

The second sample hit-test point hits two shapes: at depth 1, it hits the house rectangle; at depth 2, it hits the lawn shape. You determine which is the hit shape by specifying a depth of 1 or 2. You also specify whether QuickDraw GX returns the hit shape (by specifying level 2) or the picture that contains the hit shape (by specifying level 1).

The third hit-test point hits three shapes: at depth 1, it hits the house rectangle; at depth 2, it hits the walkway polygon; at depth 3, it hits the lawn shape. Again, you can determine which of these shapes is the hit shape (by specifying the depth) and whether the hit shape or the picture that contains it is returned (by specifying the level).

For programming examples of hit-testing picture shapes, see “Hit-Testing Pictures” beginning on page 6-46.

## Using Picture Shapes

---

This section shows you how to create, draw, edit, and hit-test picture shapes. In particular, this section shows you how to

- n create and draw pictures
- n add items to a picture
- n remove and replace items in a picture
- n provide overriding styles, inks, and transforms for the items in a picture
- n add multiple copies of a shape to a picture
- n copy objects when adding them to a picture
- n create hierarchies of pictures
- n hit-test pictures

Although the geometry of a picture shape does not contain geometric points, a picture shape can contain shapes whose geometries do contain geometric points. For this reason, some of the sample functions in this section need to specify geometric points, which are made up of two fixed-point numbers. To convert integers to fixed-point numbers when specifying geometric points, QuickDraw GX provides the `GXIntToFixed` macro:

```
#define GXIntToFixed(a) ((Fixed)(a) << 16)
```

QuickDraw GX also provides the `ff` macro as a convenient alias:

```
#define ff(a) GXIntToFixed(a)
```

The sample functions throughout this section use the `ff` macro when converting an integer constant to a fixed-point constant.

## Creating and Drawing Picture Shapes

---

QuickDraw GX provides a number of methods to create and draw pictures. In general, you can

- n define the items of the picture and draw them without creating a picture shape
- n define the items of the picture, incorporate them into a picture shape, and draw the picture shape

You can use the `GXDrawPicture` function to draw pictures using the first method. You send five parameters to this function: a count of how many shapes are in the picture, an array of references to the shapes you want drawn, and arrays of references to the overriding styles, inks, and transforms for those shapes. (See “Using Overriding Styles, Inks, and Transforms” beginning on page 6-38 for examples of overriding styles, inks, and transforms.) The `GXDrawPicture` function creates a temporary picture shape using the information in the arrays you provide, draws the picture shape, and then disposes of the temporary picture shape.

The `GXDrawPicture` function is convenient if you have a set of shapes (and overriding styles, inks, and transforms) that you want to draw only one time.

QuickDraw GX also provides a number of ways for you to create a more permanent picture shape—one that you can edit and draw repeatedly. To create a picture shape, you can

- n create an empty picture shape using the `GXNewShape` function and add items to the picture all at once using the `GXSetPicture` function
- n create an empty picture shape using the `GXNewShape` function and add items to the picture individually using the `GXSetPictureParts` function or the `AddToPicture` library function
- n create a picture with an initial set of items using the `GXNewPicture` function

In any of these three cases, you draw the picture shape using the `GXDrawShape` function.

The `GXSetPicture` function allows you to replace the entire geometry of a picture with a new set of items. For more information, see “Getting and Setting Picture Geometries” beginning on page 6-31.

The `GXSetPictureParts` function provides more sophisticated editing of a picture shape’s item list. For more information, see “Adding Items to a Picture” beginning on page 6-32, and “Removing and Replacing Items in a Picture” beginning on page 6-35.

The `GXNewPicture` function is similar to the `GXDrawPicture` function in that it requires four arrays as parameters: arrays of references to the shapes, the overriding styles, the overriding inks, and the overriding transforms that make up the items of the picture shape. However, unlike the `GXDrawPicture` function, the `GXNewPicture` function creates a picture shape and returns a reference to it to your application. You can use this reference to draw the picture using the `GXDrawShape` function.

## Picture Shapes

Listing 6-1 shows how to draw a picture of a house comprising three shapes: a rectangle for the house itself, another rectangle for the door, and a triangle for the roof. The sample function shown in this listing creates three shapes and draws them using the `GXDrawPicture` function.

---

**Listing 6-1**      Creating a simple picture of a house

```
static gxShape DrawHousePicture(void)
{
    const gxRectangle houseGeometry = {ff(90), ff(80),
                                       ff(200), ff(125)};

    const gxRectangle doorGeometry = {ff(155), ff(95),
                                       ff(170), ff(125)};

    const long roofGeometry[] = {1, /* number of contours */
                                 3, /* number of points */
                                 ff(80), ff(80),
                                 ff(145), ff(50),
                                 ff(210), ff(80)};

    gxShape houseRectangle;
    gxShape roofPolygon;
    gxShape doorRectangle;

    gxShape partsOfHouse[3];

    houseRectangle = GXNewRectangle(&houseGeometry);
    SetShapeCommonColor(houseRectangle, gxGray);

    roofPolygon = GXNewPolygons((gxPolygons *) roofGeometry);

    doorRectangle = GXNewRectangle(&doorGeometry);

    partsOfHouse[0] = houseRectangle;
    partsOfHouse[1] = roofPolygon;
    partsOfHouse[2] = doorRectangle;

    GXDrawPicture(3, partsOfHouse, nil, nil, nil);
}
```

## Picture Shapes

```

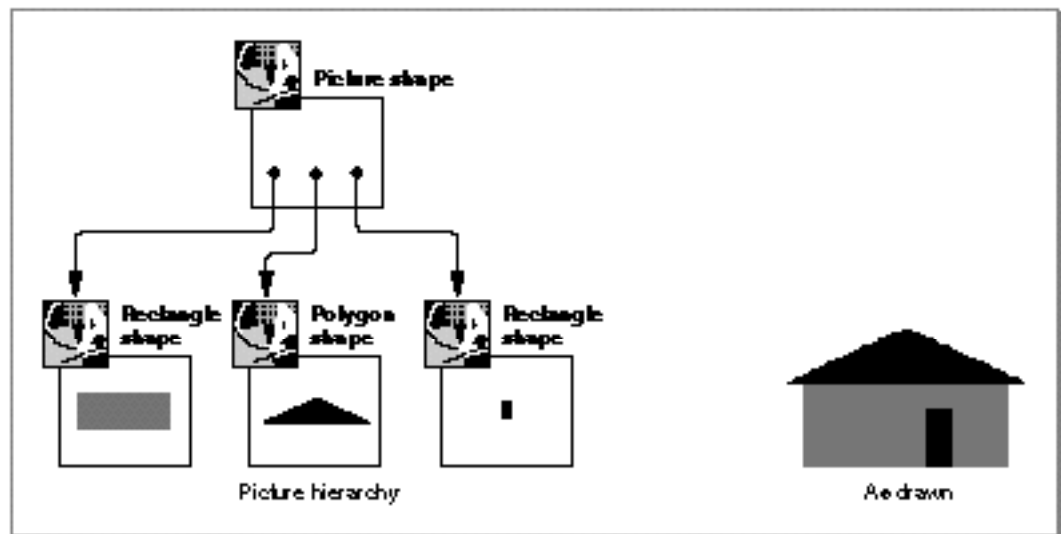
GXDisposeShape(houseRectangle);
GXDisposeShape(roofPolygon);
GXDisposeShape(doorRectangle);

};

```

The results of this sample function are shown in Figure 6-19.

**Figure 6-19** A picture of a house with a roof and a door



The call to the `GXDrawPicture` function in this example creates a temporary picture shape, draws it, and then disposes of it. This function does not return a reference to the picture shape, and so your code never has access to this shape. To create a more permanent picture shape, one that your code can reference, you must first declare a shape reference variable:

```
gxShape housePicture;
```

Then you can replace the call to the `GXDrawPicture` function with calls to the `GXNewPicture` function and the `GXDrawShape` function:

```

housePicture = GXNewPicture(3, partsOfHouse, nil, nil, nil);

GXDrawShape(housePicture);

```

The resulting picture looks the same as the picture drawn by the `GXDrawPicture` function, which is shown in Figure 6-19, but in this case the picture shape exists until you explicitly dispose of it.

## Picture Shapes

You can dispose of the picture shape using the `GXDisposeShape` function:

```
GXDisposeShape(housePicture);
```

In this example, disposing of the house picture also disposes of the three geometric shapes referenced by the house picture. That is, disposing of the house picture releases one of the references to each of the geometric shapes. However, before you dispose of the house picture, each of these shapes has an owner count of 2. (The owner count of each shape starts at 1 when you create it, and the call to the `GXNewPicture` function increments the owner count of each of the shapes.) Therefore, when you dispose of the house picture, the owner count of each of the geometric shapes decrements to 1. To free the memory used by these shapes, you must still dispose of them individually—just as you created them:

```
GXDisposeShape(houseRectangle);
GXDisposeShape(roofPolygon);
GXDisposeShape(doorRectangle);
```

Notice that you can dispose of these three geometric shapes before you dispose of the house picture, as shown in Listing 6-2.

---

**Listing 6-2** Disposing of shapes contained in a picture before disposing of the picture

```
housePicture = GXNewPicture(3, partsOfHouse, nil, nil, nil);

GXDisposeShape(houseRectangle);
GXDisposeShape(roofPolygon);
GXDisposeShape(doorRectangle);

GXDrawPicture(housePicture);

GXDisposeShape(housePicture);
```

In this example, disposing of the three geometric shapes decrements their owner count by 1, but does not free their memory because the house picture shape still contains a reference to each of the three shapes. Only when the house picture is disposed of is the memory occupied by these three geometric shapes freed.

For information about the `GXDrawPicture` function, see page 6-67. For information about the `GXNewPicture` function, see page 6-57.



## Getting and Setting Picture Geometries

---

QuickDraw GX provides the `GXGetPicture` function and the `GXSetPicture` function to allow you to examine and replace the entire geometry of a picture shape.

The `GXGetPicture` function returns as its function result the number of items in the picture, and optionally returns an array of references to the shapes referenced by the picture's items, as well as arrays of references to the picture items' overriding styles, inks, and transforms. Typically, you call this function twice. The first time, you determine the number of items in the picture. Then you use that number to allocate enough memory to hold the arrays of references. Finally, you call the function a second time to copy references from the items of the picture into your arrays.

The `GXSetPicture` function allows you to replace the geometry of a picture with a new set of items. This function increments the owner counts of the new shapes, overriding styles, overriding inks, and overriding transforms and disposes of the original shapes, overriding styles, overriding inks, and overriding transforms.

Listing 6-3 gives an example of the `GXGetPicture` function. This example, which builds on the example from Listing 6-1 on page 6-28, edits the picture of the house by moving the location of the door.

---

**Listing 6-3**      Extracting and editing items from a picture

```
gxShape  *extractedShapes;

long      numberOfItems;
.
.
.
numberOfItems = GXGetPicture(housePicture, nil, nil, nil, nil);
extractedShapes = (gxShape *)
                  NewPtr(numberOfItems * sizeof(gxShape));
GXGetPicture(housePicture, extractedShapes, nil, nil, nil);

GXMoveShape(extractedShapes[2], ff(-40), 0);

GXDrawShape(housePicture);
```

The code in Listing 6-3 includes two new variable declarations: a pointer to shape references and a long integer. The code in this listing calls the `GXGetPicture` function to determine the number of items in the house picture, uses that number to allocate enough memory to store the appropriate number of shape references, and then calls the `GXGetPicture` function a second time to copy the shape references from the items of the picture into the array of shape references. The sample code then uses the `GXMoveShape` function to move the third shape in the picture 40 grid points to the left. Notice that the `extractedShapes` array does not contain copies of the shapes in the picture; instead, it contains copies of references to the shapes in the picture. The references in the `extractedShapes` array reference the actual shapes in the picture. Therefore, moving the shape referenced by the third item in the `extractedShapes` array actually affects the house picture, as shown in Figure 6-20.

**Figure 6-20** A picture of a house with a relocated door



For more information about the `GXGetPicture` and `GXSetPicture` functions, see page 6-59 through page 6-63.

## Adding Items to a Picture

Once you have created a picture shape, you can add more items to it using one of these methods:

- n You can use the `GXGetPicture` function to obtain arrays of references to the shapes, overriding styles, overriding inks, and overriding transforms that make up the items of a picture. You can then add new references to these arrays and use the `GXSetPicture` function to replace the original items with the information in the edited arrays.
- n You can use the `GXSetPictureParts` function to insert any number of new items directly into a picture shape. With this function, you can insert the new items anywhere in the existing item list.
- n You can use the `AddToPicture` library function to insert a single new item at the end of a picture shape's item list.

Listing 6-4 and Listing 6-5 show how to use the `GXSetPictureParts` function to add three new items to the house picture defined in Listing 6-1 on page 6-28. Listing 6-4 defines three new shapes to include in the picture, and Listing 6-5 uses the `GXSetPictureParts` function to insert the shapes into the house picture.

---

**Listing 6-4**      Defining new shapes for the house picture

```

gxShape  lawnPolygon;
gxShape  walkwayPolygon;
gxShape  chimneyRectangle;
.
.
.
const long lawnGeometry[] = {1, /* number of contours */
                             5, /* number of points */
                             0x70000000, /* 0111 0000 ... */
                             ff(20), ff(160), /* on */
                             ff(20), ff(130), /* off */
                             ff(140), ff(100), /* off */
                             ff(260), ff(130), /* off */
                             ff(260), ff(160)}; /* on */

const long walkwayGeometry[] = {1, /* number of contours */
                                3, /* number of points */
                                ff(102), ff(160),
                                ff(122), ff(100),
                                ff(142), ff(160)};

gxRectangle chimneyGeometry = {ff(110), ff(50),
                               ff(120), ff(80)};

lawnPolygon = GXNewPaths((gxPaths *) lawnGeometry);
SetShapeCommonColor(lawnPolygon, light + gxGray);

walkwayPolygon = GXNewPolygons((gxPolygons *) walkwayGeometry);
SetShapeCommonColor(walkwayPolygon, dark + gxGray);

chimneyRectangle = GXNewRectangle(&chimneyGeometry);
SetShapeCommonColor(chimneyRectangle, dark + gxGray);

```

## Picture Shapes

The sample code from Listing 6-4 defines a lawn shape, a walkway shape, and a chimney shape. The sample code in Listing 6-5 creates an array to store references to these three shapes, and then calls the `GXSetPictureParts` function to insert the shapes into the house picture.

---

**Listing 6-5** Adding new shapes to the house picture

```

gxShape  insertedShapes[3];
.
.
.
insertedShapes[0] = lawnPolygon;
insertedShapes[1] = walkwayPolygon;
insertedShapes[2] = chimneyRectangle;

GXSetPictureParts(housePicture,
                  1, /* insert before first item */
                  0, /* don't replace any existing items */
                  3, /* insert three new items */
                  insertedShapes, /* shapes to insert */
                  nil, nil, nil); /* no overrides */

```

The first parameter to the `GXSetPictureParts` function specifies the picture whose item list you want to edit. The second parameter specifies where you want the editing to occur. In this example, the second parameter is set to 1, which indicates that the new items should be inserted before the first item of the picture. QuickDraw GX draws the items of a picture in order from back to front; therefore, inserting the new items before the existing items ensures that the new items are drawn behind the existing ones.

The third parameter to the `GXSetPictureParts` function specifies how many of the original picture items to remove. In this example, this parameter is set to 0. For examples of removing and replacing picture items, see the next section.

The fourth parameter to the `GXSetPictureParts` function specifies how many new items to insert into the picture, which in this case is 3.

The last four parameters to the `GXSetPictureParts` function specify the shapes, overriding styles, overriding inks, and overriding transforms that make up the new picture items.

## Picture Shapes

Once you have inserted the new shapes into the picture, you can dispose of the shapes (which simply lowers their owner count to 1), and then draw the picture:

```
GXDisposeShape(lawnPolygon);  
GXDisposeShape(walkwayPolygon);  
GXDisposeShape(chimneyRectangle);  
  
GXDrawShape(housePicture);
```

The resulting picture is shown in Figure 6-21.

---

**Figure 6-21** A house with a lawn, walkway, and chimney



For more information about the `GXSetPictureParts` function, see page 6-65.

## Removing and Replacing Items in a Picture

---

You can use the `GXSetPicture` function or the `GXSetPictureParts` function to replace items in a picture.

The `GXSetPicture` function removes every item in a picture and inserts a new list of items. The `GXSetPictureParts` function allows you more control in replacing items. With this function, you can replace a subset of the items in a picture with another set of items. The inserted set does not have to have the same number of items as the replaced set.

As a simple example, you can use the `GXSetPictureParts` function to remove a single item from a picture. Listing 6-6 shows how to use the `GXSetPictureParts` function to remove the chimney, which is item number 3, from the house picture shown in Figure 6-21.

---

**Listing 6-6** Removing an item from a picture

```
GXSetPictureParts(housePicture,  
                 3, /* start editing at item 3 */  
                 1, /* remove 1 item */  
                 0, /* insert 0 items */  
                 nil, /* no shapes to insert */  
                 nil, nil, nil); /* no overrides */
```

The resulting picture is shown in Figure 6-22.

---

**Figure 6-22** A house with chimney removed



You can also use the `GXSetPictureParts` function to replace items in a picture; with a single call to `GXSetPictureParts`, you can remove items from a picture and insert new items into a picture.

Like the sample code in Listing 6-6, the sample code in Listing 6-7 uses the `GXSetPictureParts` function to remove the chimney shape from the house picture. However, this call to the `GXSetPictureParts` function inserts a new chimney into the house picture at the same time.

---

**Listing 6-7** Replacing one shape with another

```
gxShape newChimneyRectangle;

gxRectangle newChimneyGeometry = {ff(170), ff(50),
                                   ff(180), ff(80)};

.
.
.
newChimneyRectangle = GXNewRectangle(&newChimneyGeometry);
SetShapeCommonColor(newChimneyRectangle, dark + gxGray);

GXSetPictureParts(housePicture,
                  3, /* start editing at item 3 */
                  1, /* remove 1 item */
                  1, /* insert 1 item */
                  &newChimneyRectangle, /* shape to insert */
                  nil, nil, nil); /* no overrides */

GXDisposeShape(newChimneyRectangle);
```

The resulting house picture is shown in Figure 6-23.

---

**Figure 6-23** A house with the chimney replaced



For more information about the `GXSetPictureParts` function, see page 6-65.

## Using Overriding Styles, Inks, and Transforms

---

As detailed in the previous three sections, QuickDraw GX provides a number of methods for adding items to a picture shape. In particular, you can add items when creating a picture using the `GXNewPicture` function, you can replace every item in an existing picture using the `GXSetPicture` function, and you can replace some of the items in a picture using the `GXSetPictureParts` function. All three of these functions allow you to specify overriding styles, inks, and transforms for the new picture items.

As an example, the code in Listing 6-8 and Listing 6-9 alters the house picture from Listing 6-1 on page 6-28. Listing 6-8 defines a style object, an ink object, and a transform object. Listing 6-9 uses these objects and the `GXSetPicture` function to create a house picture whose items contain overriding styles, inks, and transforms.

---

**Listing 6-8**      Creating style, ink, and transform objects

```

gxShape squarePattern;
gxStyle patternedStyle;
gxInk grayInk;
gxTransform skewedTransform;

const gxRectangle squareGeometry = {ff(0), ff(0),
                                     ff(2), ff(2)};

gxPatternRecord patternRecord;
.
.
.
squarePattern = GXNewRectangle(&squareGeometry);
patternRecord.attributes = gxNoAttributes;
patternRecord.pattern = squarePattern;
patternRecord.u.x = ff(1);
patternRecord.u.y = ff(4);
patternRecord.v.x = ff(3);
patternRecord.v.y = ff(1);

patternedStyle = GXNewStyle();
GXSetStylePattern(patternedStyle, &patternRecord);

grayInk = GXNewInk();
SetInkCommonColor(grayInk, gxGray);

skewedTransform = GXNewTransform();
GXSkewTransform(skewedTransform, -fl(.5), 0, ff(122), ff(110));

```



## Picture Shapes

Listing 6-9 uses the style, ink, and transform objects defined in Listing 6-8, and the `partsOfHouse` array (which is defined in Listing 6-1 on page 6-28) to create a house picture. In this house picture, the main part of the house has an overriding style, the roof has an overriding ink, and the door has an overriding transform.

---

**Listing 6-9**      Creating a picture whose items have overriding styles, inks, and transforms

```

gxStyle  overridingStyles[3];
gxInk    overridingInks[3];
gxTransform overridingTransforms[3];
.
.
.
overridingStyles[0] = patternedStyle;
overridingStyles[1] = nil;
overridingStyles[2] = nil;

overridingInks[0] = nil;
overridingInks[1] = grayInk;
overridingInks[2] = nil;

overridingTransforms[0] = nil;
overridingTransforms[1] = nil;
overridingTransforms[2] = skewedTransform;

housePicture = GXNewShape(gxPictureType);
GXSetPicture(housePicture,
             3,
             partsOfHouse,
             overridingStyles,
             overridingInks,
             overridingTransforms);

```

Once you have added the overriding style, ink, and transform objects to the picture, you can dispose of them, as shown in Listing 6-10. Since these objects are referenced twice (once by your application and once by the house picture), disposing of them lowers their owner counts to 1, but does not free the memory associated with them. When you eventually dispose of the house picture, QuickDraw GX disposes of these objects again and frees their memory.

---

**Listing 6-10** Disposing of overriding style, ink, and transform objects before drawing

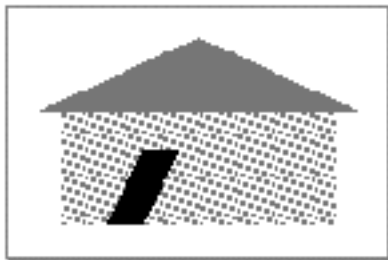
```
GXDisposeShape(squarePattern);
GXDisposeStyle(patternedStyle);
GXDisposeInk(grayInk);
GXDisposeTransform(skewedTransform);

GXDrawShape(housePicture);
```

The resulting picture is shown in Figure 6-24.

---

**Figure 6-24** A house picture with an overriding style, ink, and transform



For more information about overriding styles, inks, and transforms, see “Overriding Styles, Inks, and Transforms” beginning on page 6-8.

For more information about the `GXNewPicture` function, see page 6-57. For more information about the `GXSetPicture` function, see page 6-61.

## Adding Multiple References

---

Multiple items in a single picture can reference the same shape. You can use any of the functions that add items to a picture (`GXNewPicture`, `GXSetPicture`, `GXSetPictureParts`) to add multiple references to a single shape. The example in Listing 6-11 adds four new items to the house picture defined in Listing 6-1 on page 6-28. Each of these items references the same shape—a small, white rectangle. Because four items reference the same rectangle, four instances of this rectangle appear in the picture.

## Picture Shapes

Without overriding transforms, however, all four instances of this rectangle would appear in the same location. Therefore, the sample code in Listing 6-11 creates overriding transforms for three of the four new items.

---

**Listing 6-11** Adding four items that reference the same shape to a house picture

```

gxRectangle windowGeometry = {ff(155), ff(93),
                               ff(160), ff(112)};

gxShape windowRectangle;

gxShape insertedShapes[4];
gxTransform overridingTransforms[4];

windowRectangle = GXNewRectangle(&windowGeometry);
SetShapeCommonColor(windowRectangle, gxWhite);

insertedShapes[0] = windowRectangle;
insertedShapes[1] = windowRectangle;
insertedShapes[2] = windowRectangle;
insertedShapes[3] = windowRectangle;

overridingTransforms[0] = nil;
overridingTransforms[1] = GXNewTransform();
overridingTransforms[2] = GXNewTransform();
overridingTransforms[3] = GXNewTransform();

GXMoveTransform(overridingTransforms[1], ff(7), 0);
GXMoveTransform(overridingTransforms[2], ff(14), 0);
GXMoveTransform(overridingTransforms[3], ff(21), 0);

GXSetPictureParts(housePicture,
                  3, /* where to insert */
                  0, /* how many to replace */
                  4, /* how many to insert */
                  insertedShapes,
                  nil, nil,
                  overridingTransforms);

```

This sample code creates one rectangle shape and three transform objects. Once you insert these objects in the picture, you can dispose of them to lower their owner count to 1, as shown in Listing 6-12. Since these objects are referenced twice (once by your application and once by the house picture), disposing of them lowers their owner counts to 1, but does not free the memory associated with them. When you dispose of the house picture, QuickDraw GX disposes of these objects again and frees their memory.

---

**Listing 6-12** Disposing of the white rectangle and the three transform objects before drawing

```
int count;
.
.
.
for (count = 1; count <= 3 ; count++)
    GXDisposeTransform(overridingTransforms[count]);

GXDisposeShape(windowRectangle);

GXDrawShape(housePicture);
```

The resulting picture is shown in Figure 6-25.

---

**Figure 6-25** A house with four windows



Notice that the sample code in Listing 6-11 creates three separate transform objects because three different transformations are happening to the instances of the window rectangle—the second instance is moved 7 grid points to the right, the third instance is moved 14 grid points to the right, and the fourth instance is moved 21 grid points to the right.

You can specify that QuickDraw GX copy the overriding transforms when adding them to the picture (rather than adding them by reference) by setting the unique items shape attribute, as discussed in the next section.

For more information about adding multiple items referencing the same shape, see “Multiple References” beginning on page 6-10.

## Adding Items With the Unique Items Attribute Set

---

The unique items shape attribute changes the way in which QuickDraw GX adds shapes to a picture. When you add a shape to a picture that does not have this attribute set, QuickDraw GX copies the reference to the existing shape, inserts this reference into the picture's item list, and increments the owner count of the shape. Similarly, if you specify an overriding style, ink, or transform object for the shape, QuickDraw GX copies the object's reference into the picture's item list and increments the owner count of the object.

However, when you add a shape to a picture that has the unique items attribute set, QuickDraw GX makes a copy of the shape and inserts a reference to the copy in the picture's item list. Similarly, overriding styles, inks, and transforms are also copied.

As an example, Listing 6-13 shows how to use the `GXGetShapeAttributes` and `GXSetShapeAttributes` functions to set the unique items shape attribute of a picture. You must set this attribute before you add any items to a picture; if the picture already contains items, setting this attribute results in an error.

This listing adds four instances of a window rectangle to the house picture from Listing 6-1 on page 6-28. This sample code specifies the same overriding transform for each instance of the window rectangle. However, the overriding transform is moved (with the `GXMoveTransform` function) after each call to the `AddToPicture` library function. Because the house picture has the unique items shape attribute set, QuickDraw GX makes a separate copy of the overriding transform each time a window rectangle is inserted into the picture.

---

**Listing 6-13** Adding unique items to a picture

```
GXSetShapeAttributes(housePicture,
    GXGetShapeAttributes(housePicture) | gxUniqueItemsShape);
.
.
.
moveToRight = GXNewTransform();

for (count = 0; count <= 3 ; count++) {
    AddToPicture(housePicture,
        windowRectangle,
        nil, nil,
        moveToRight);

    GXMoveTransform(moveToRight, ff(7), 0);
}
```

In this example, the first time that the `AddToPicture` function is called, QuickDraw GX creates a copy of the window rectangle shape and a copy of the overriding transform object, and inserts references to the copies in the item list of the house picture.

## Picture Shapes

The second time that the `AddToPicture` function is called, QuickDraw GX notices that the window rectangle shape has not changed, so it does not make another copy of the window rectangle. Instead, it creates a new item in the house picture that references the previously made copy. However, the overriding transform has changed, so QuickDraw GX makes a new copy of it for the new picture item.

The third and fourth calls to the `AddToPicture` function also create new copies of the overriding transform, but do not create new copies of the window rectangle.

After the code from Listing 6-13 finishes executing, there are a total of two window rectangles—the original one, which is referenced by the `windowRectangle` variable, and the copy, which is referenced four times by the items of the house picture. There are a total of five transform objects—the original one, which is referenced by the `moveToRight` variable, and four separate copies referenced by the four new items of the picture.

Figure 6-26 shows the resulting picture.

---

**Figure 6-26** A house with four windows and four unique overriding transforms



For more information about the unique items shape attribute, see “Unique Items Shape Attribute” beginning on page 6-15.

## Creating Picture Hierarchies

---

QuickDraw GX allows the items in a picture shape to reference other picture shapes. You can use any of the functions that allow you to add items to pictures (`GXNewPicture`, `GXSetPicture`, `GXSetPictureParts`) to create picture hierarchies.

When drawing a picture hierarchy, QuickDraw GX concatenates the mapping and clipping information contained in the transform objects (or overriding transform objects) at each level of the hierarchy. As an example, Listing 6-14 shows how QuickDraw GX concatenates mapping information from two levels of a picture hierarchy. In this example, the house picture from Figure 6-21 on page 6-35 is added to another picture as an item with an overriding transform that rotates the house clockwise 90 degrees. In turn, this picture is added as an item to yet another picture, with the same overriding transform.

**Listing 6-14** Creating a picture hierarchy

---

```

gxShape rootPicture, level1Picture;
gxTransform rotateHouse;
.
.
.
rotateHouse = GXNewTransform();
GXRotateTransform(rotateHouse, ff(90), ff(150), ff(100));

level1Picture = GXNewPicture(1,
                             &housePicture,
                             nil, nil,
                             &rotateHouse);
rootPicture = GXNewPicture(1,
                           &level1Picture,
                           nil, nil,
                           &rotateHouse);

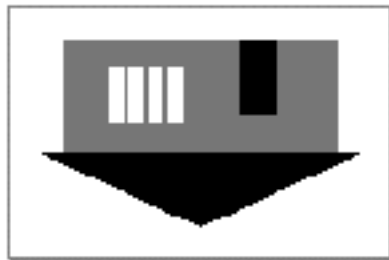
GXDrawShape(rootPicture);

```

When QuickDraw GX draws the root-level picture, it concatenates the information in the two overriding transforms, and draws the house picture rotated clockwise 180 degrees, as shown in Figure 6-27.

---

**Figure 6-27** A house rotated by 90 degrees two times



For more information about picture hierarchies and transform concatenation, see “Picture Hierarchies” beginning on page 6-18 and “Transform Concatenation” beginning on page 6-19.

## Hit-Testing Pictures

---

As described in “About Hit-Testing Picture Shapes” beginning on page 6-24, QuickDraw GX hit-tests a picture shape by

- n hit-testing each item contained in the picture, using the hit-test information in that item’s transform object (or overriding transform object, if the item has one) to determine if the item was hit or not
- n finding the hit item that corresponds to the depth you specify
- n determining the item to return using the level you specify
- n providing information about the item

The criteria you specify includes the depth at which you want to hit-test the picture, and the level of the picture hierarchy at which you want to hit-test.

To illustrate picture hit-testing, Listing 6-15 creates a picture hierarchy using the shapes defined in Listing 6-1 on page 6-28 and Listing 6-4 on page 6-33. This example creates a picture shape that contains two items. The first item is a picture of a lawn and a walkway, and the second item is a picture of a house, roof, and door.

---

**Listing 6-15**    Creating a picture hierarchy

```
gxShape  groundsPicture, housePicture, entirePicture;

gxShape  partsOfHouse[4];
gxShape  partsOfGrounds[2];
gxShape  partsOfEntirePicture[2];
.
.
.
partsOfGrounds[0] = lawnPolygon;
partsOfGrounds[1] = walkwayPolygon;
groundsPicture = GXNewPicture(2, partsOfGrounds, nil, nil, nil);

partsOfHouse[0] = houseRectangle;
partsOfHouse[1] = roofPolygon;
partsOfHouse[2] = doorRectangle;
housePicture = GXNewPicture(3, partsOfHouse, nil, nil, nil);

partsOfEntirePicture[0] = groundsPicture;
partsOfEntirePicture[1] = housePicture;
entirePicture = GXNewPicture(2, partsOfEntirePicture,
                             nil, nil, nil);
```



# Picture Shapes

Figure 6-28 shows the items that make up the grounds picture.

**Figure 6-28** Grounds picture

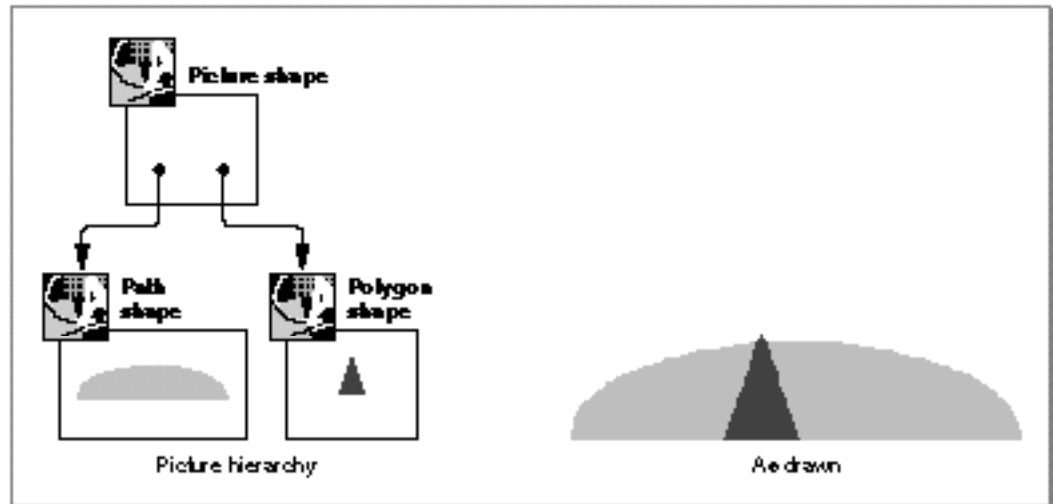


Figure 6-29 shows the items that make up the house picture.

**Figure 6-29** House picture

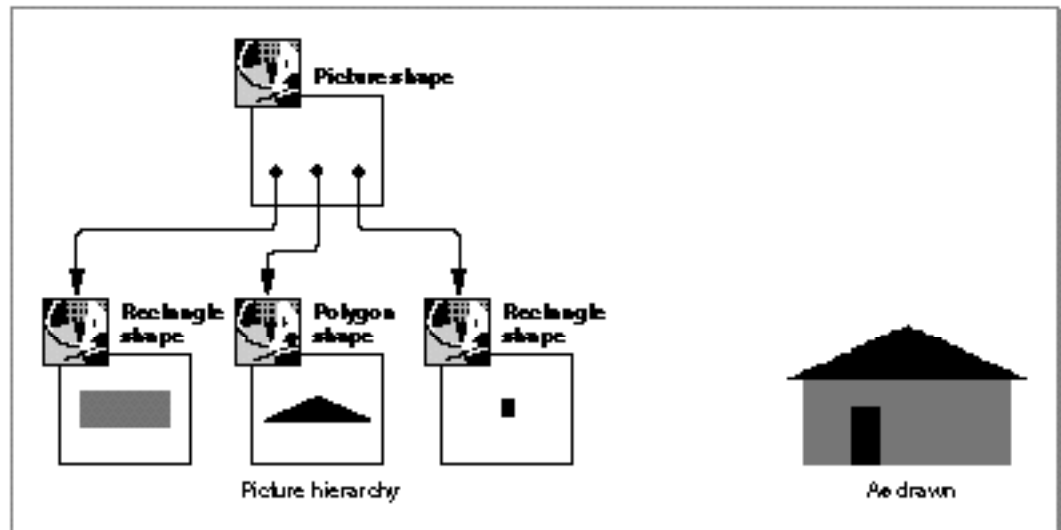
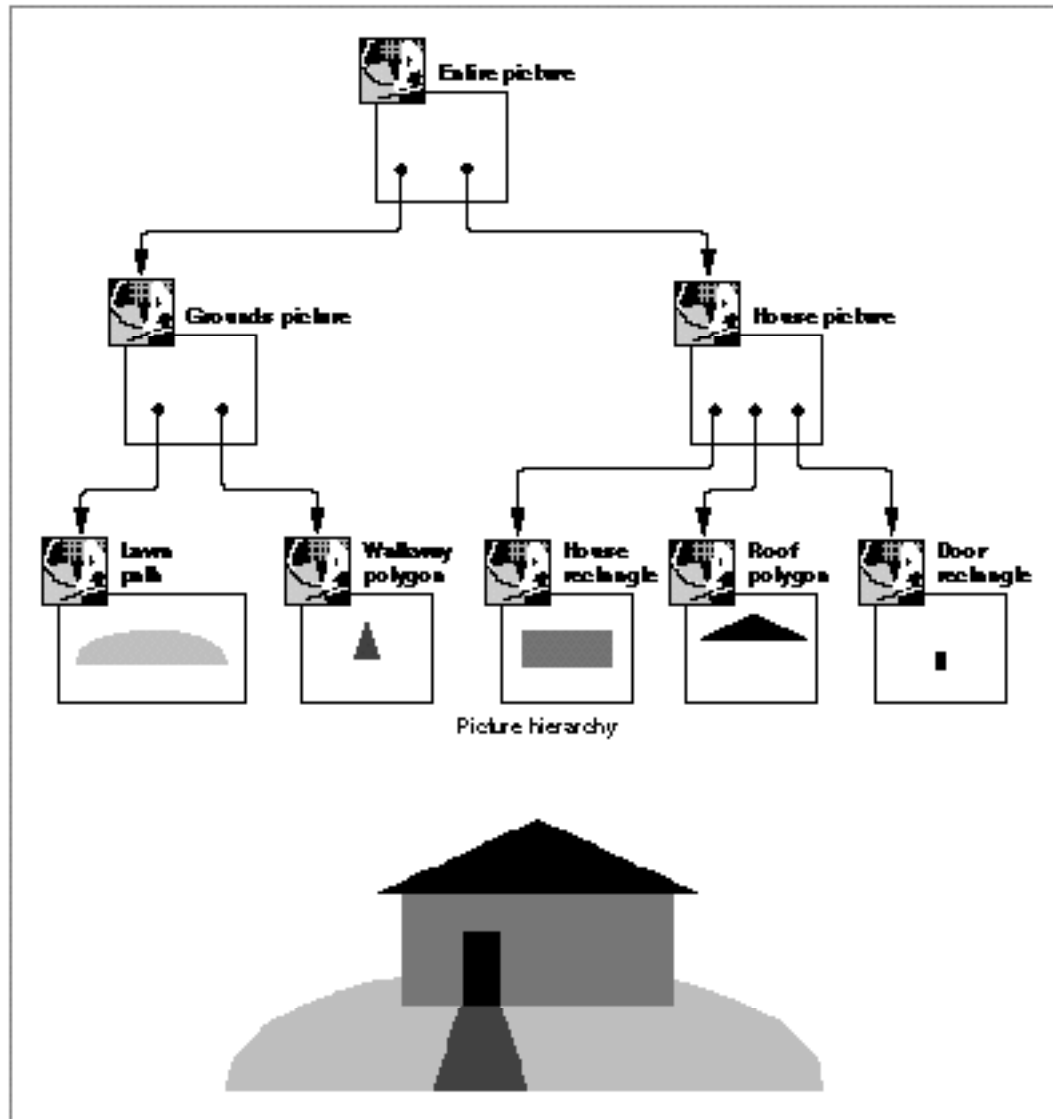


Figure 6-30 shows the entire picture created in Listing 6-15.

**Figure 6-30** Picture containing grounds picture and house picture



You hit-test a picture shape using the function `GXHitTestPicture`. This function takes as its parameters a reference to the picture to hit-test, the test point, an optional hit-test parameters structure, the level at which to hit-test, and the depth at which to hit-test. The sample code in Listing 6-16 shows how to hit-test the picture from Listing 6-15 using a test point of `ff(122), ff(110)`.

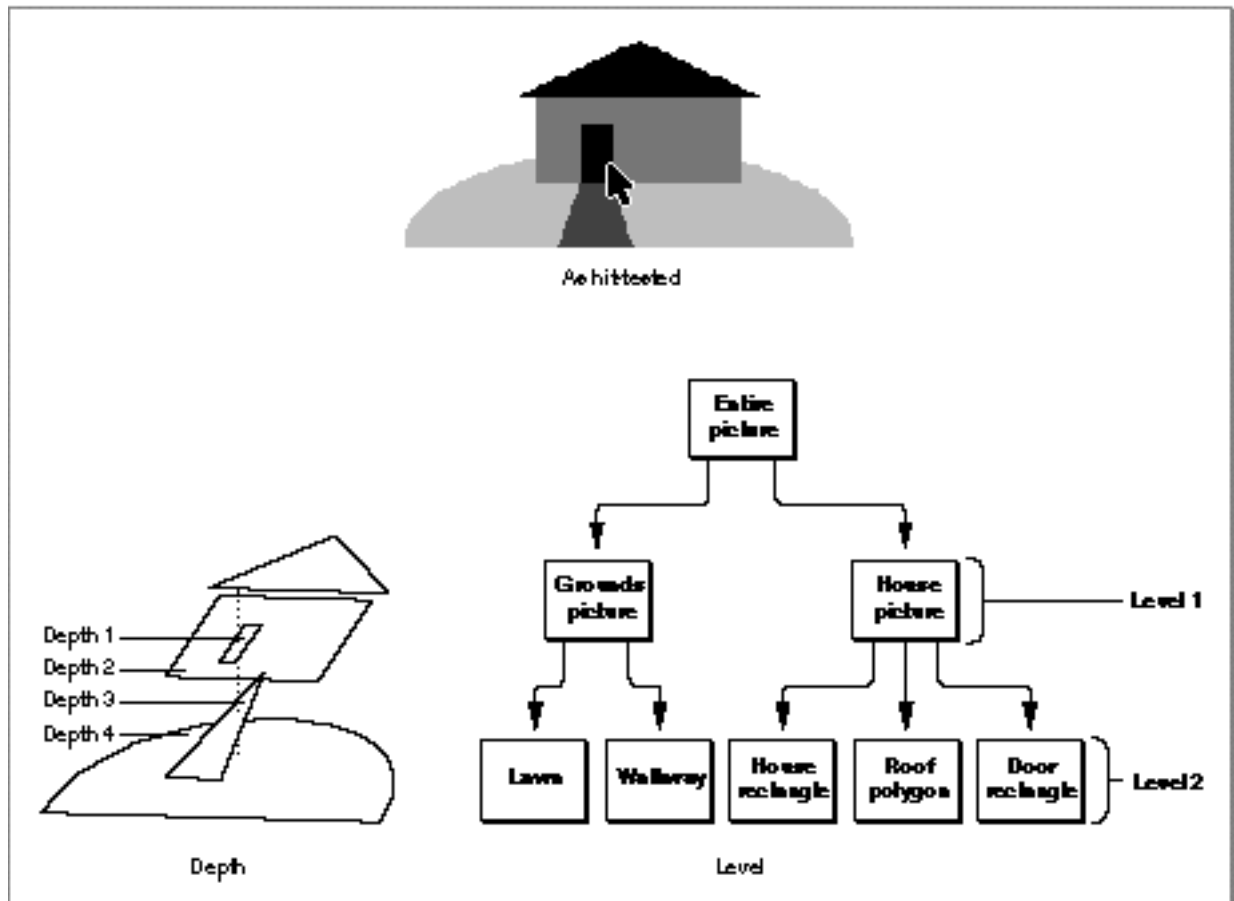
**Listing 6-16** Hit-testing a picture shape

```

gxPoint testPoint = {ff(122), ff(110)};
gxShape hitShape;
long level, depth;
.
.
.
hitShape = GXHitTestPicture(entirePicture, &testPoint, nil,
                             level, depth);

```

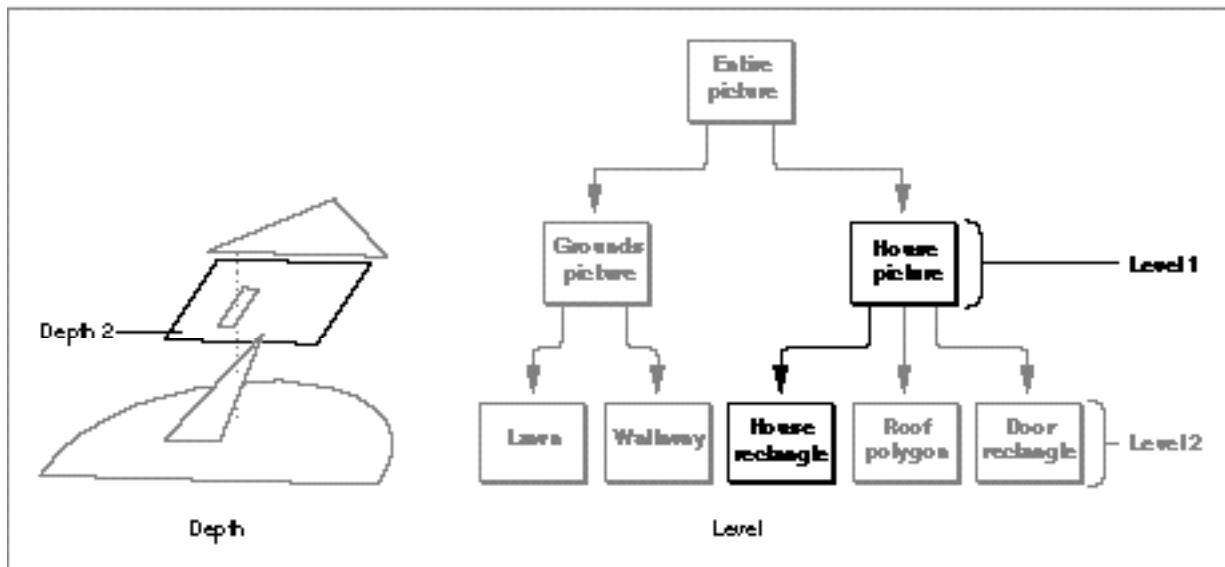
Figure 6-31 shows the location of the test point.

**Figure 6-31** Hit-testing the picture of house and grounds

## Picture Shapes

If you specify a depth of 2, the hit shape is the house rectangle. If you specify a level of 1, QuickDraw GX returns information about the house picture that contains the house rectangle. Figure 6-32 depicts this selection process.

**Figure 6-32** Hit-testing the picture at depth 2 and level 1



The `GXHitTestPicture` function returns a reference to the shape that was hit by the test point. In this example, the test point falls above four separate shapes: the door rectangle, the house rectangle, the walkway polygon, and the lawn path. By varying the values of the level and depth parameters, you can control which shape is returned by the `GXHitTestPicture` function.

Table 6-1 shows which shape is returned for various choices of level and depth.

**Table 6-1** Hit-testing a picture at different depths and levels

Depth	Level	Hit Shape
1	2	Door rectangle
1	1	House picture
2	2	House rectangle
2	1	House picture
3	2	Walkway polygon
3	1	Grounds picture
4	2	Lawn path
4	1	Grounds picture

At depth 1, the returned shape is the frontmost shape that was hit—in this case, the door rectangle, which is at level 2 in the picture hierarchy. If you specify a depth of 1 and a level of 1, the `GXHitTestPicture` function returns the picture that contains the door rectangle—in this case the house picture.

In a similar manner, depth 2 indicates the house rectangle, depth 3 indicates the walkway polygon, and depth 4 indicates the lawn path.

For information about the `GXHitTestPicture` function, see page 6-69.

## Applying Functions Described Elsewhere to Picture Shapes

---

QuickDraw GX provides only a small number of functions that apply exclusively to picture shapes. However, many of the QuickDraw GX functions that you can apply to other types of shapes you can also apply to picture shapes.

The next six sections discuss how functions described elsewhere operate when applied to picture shapes. These sections are

- n “Functions That Post Errors or Warnings When Applied to Pictures” on page 6-52, which lists functions that you can apply to other types of shapes but not to picture shapes
- n “Shape-Related Functions Applicable to Pictures” on page 6-54, which lists functions that operate on picture shape objects
- n “Geometric Operations Applicable to Pictures” on page 6-55, which lists the few geometric operation functions that you can apply to pictures
- n “Style-Related Functions Applicable to Pictures” on page 6-55, which discusses how style-related functions apply to pictures
- n “Ink-Related Functions Applicable to Pictures” on page 6-56, which discusses how ink-related functions apply to pictures
- n “Transform-Related Functions Applicable to Pictures” on page 6-56, which discusses how transform-related functions apply to pictures

### Functions That Post Errors or Warnings When Applied to Pictures

---

Some QuickDraw GX functions that operate on other types of shapes do nothing but post an error or a warning if you try to apply them to a picture shape.

For example, there are a number of shape-related functions and geometric operations that you cannot apply to picture shapes. Table 6-2 lists these functions, which are described in full in Chapter 2, “Geometric Shapes,” and Chapter 4, “Geometric Operations.”

**Table 6-2** Geometric operations that post errors or warnings when applied to pictures

Function name	Error or warning posted
GXBreakShape	graphic_type_does_not_contain_points
GXContainsShape	shape_operator_may_not_be_a_picture
GXCountShapePoints	graphic_type_does_not_contain_points
GXDifferenceShape	shape_operator_may_not_be_a_picture
GXExcludeShape	shape_operator_may_not_be_a_picture
GXGetShapeCenter	illegal_type_for_shape
GXGetShapeDirection	graphic_type_does_not_have_multiple_contours
GXGetShapeLength	shape_does_not_have_length
GXGetShapePoints	graphic_type_does_not_contain_points
GXInsetShape	graphic_type_cannot_be_inset
GXIntersectShape	shape_operator_may_not_be_a_picture
GXInvertShape	shape_cannot_be_inverted
GXReverseDifferenceShape	shape_operator_may_not_be_a_picture
GXReverseShape	contour_out_of_range
GXShapeLengthToPoint	shape_does_not_have_length
GXSetShapePoints	graphic_type_does_not_contain_points
GXTouchesShape	shape_operator_may_not_be_a_picture
GXUnionShape	shape_operator_may_not_be_a_picture

Most of these geometric operations do not apply to picture shapes because a picture's geometry is substantially different from the geometry of a geometric shape.

You can apply a few of the geometric operations to pictures, however. These functions are discussed in “Geometric Operations Applicable to Pictures” beginning on page 6-55.

## Shape-Related Functions Applicable to Pictures

You can apply all of the functions described in the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects* to picture shapes. These functions allow you to

- n manipulate the shape object that represents the picture shape; for example, you can copy, clone, cache, compare, and dispose of the picture shape
- n set the geometry, shape type, shape fill, and shape attributes of the picture shape
- n change the style, ink, and transform objects that are associated with the picture shape
- n manipulate the picture shape’s tags and owner count

Table 6-3 gives important picture-related information for a subset of the functions from the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects*. Functions described in that chapter that do not appear in this list exhibit the same behavior when applied to picture shapes as they do when applied to other types of shapes.

**Table 6-3** Shape-related functions that exhibit special behavior when applied to pictures

Function name	Action taken
GXCopyToShape	Makes a copy of the picture shape; the items of the new picture shape reference the same shapes as the items of the original picture shape.
GXCopyDeepToShape	Makes a copy of the picture shape, including a complete copy of the entire picture hierarchy.
GXGetShapeSize	Determines the amount of memory currently used by all of the items in the picture.
GXGetShapeFill	Returns the shape’s shape fill, which for picture shapes is always even-odd fill or no fill.
GXSetShapeFill	Sets the shape’s shape fill; you must always set a picture shape’s shape fill to even-odd fill or no fill.
GXSetShapeType	Changes the shape type of the picture shape and converts the shape fill and geometry as appropriate. The resulting shape is a picture with one item—the original shape.



## Geometric Operations Applicable to Pictures

---

Many geometric operations post errors or warnings when applied to picture shapes, as described in “Functions That Post Errors or Warnings When Applied to Pictures” on page 6-52.

You can, however, apply the remainder of the functions described in Chapter 4, “Geometric Operations,” to picture shapes. Table 6-4 gives important picture-related information for a subset of these functions; the remainder of the geometric operations exhibit the same behavior when applied to picture shapes as they do when applied to other types of shapes.

**Table 6-4** Geometric operations that exhibit special behavior when applied to pictures

---

Function name	Action taken
<code>GXGetShapeArea</code>	Returns summed areas of picture items.
<code>GXGetShapeBounds</code>	Returns bounding rectangle of specified item.
<code>GXSetShapeBounds</code>	If the picture’s <code>mapTransformShape</code> shape attribute is set, this function changes the picture’s transform so that the entire picture fits within the specified bounding rectangle. If this attribute is not set, this function posts an error.

## Style-Related Functions Applicable to Pictures

---

Picture shapes make limited use of their style objects. You may apply to a picture shape any of the functions described in Chapter 3, “Geometric Styles,” (such as `GXSetShapePen`, `GXSetShapeDash`, and so on) to set the properties of a picture’s style object, and you may use the corresponding functions (`GXGetShapePen`, `GXGetShapeDash`, and so on) to examine these properties. However, `QuickDraw GX` ignores these properties when drawing a picture.

## Ink-Related Functions Applicable to Pictures

---

Picture shapes make limited use of their ink objects. You may apply to a picture shape any of the shape-related functions described in the chapter “Ink Objects” of *Inside Macintosh: QuickDraw GX Objects* (such as `GXSetShapeColor`, `GXSetShapeTransfer`) to set the properties of a picture’s ink object, and you may use the corresponding functions (`GXGetShapeColor`, `GXGetShapeTransfer`) to examine these properties. However, QuickDraw GX ignores these properties when drawing a picture.

## Transform-Related Functions Applicable to Pictures

---

Although picture shapes do not make full use of their style and ink objects, they do make full use of their transform objects. You can apply all of the shape-related functions that are described in the chapter “Transform Objects” of *Inside Macintosh: QuickDraw GX Objects* to picture shapes.

In general, you need to be sure that a picture shape’s `gxMapTransformShape` shape attribute is set before applying any of the mapping operations to a picture.

## Picture Shapes Reference

---

### Functions

---

This section describes the functions provided by QuickDraw GX specifically for creating and manipulating picture shapes. With the functions described in this section, you can

- n create a new picture shape
- n examine and edit the items of a picture shape
- n draw pictures
- n hit-test pictures

See the section “Applying Functions Described Elsewhere to Picture Shapes” beginning on page 6-52 for information about other QuickDraw GX functions that you can apply to picture shapes.

### Creating Picture Shapes

---

This section describes the `GXNewPicture` function, which you use to create new picture shapes.

### GXNewPicture

---

You can use the `GXNewPicture` function to create a new picture shape.

```
gxShape GXNewPicture(long count, const gxShape shapes[],
                     const gxStyle styles[], const gxInk inks[],
                     const gxTransform transforms[])
```

<code>count</code>	The number of picture items in the new picture shape.
<code>shapes</code>	An array of references to the shapes you want to include in the picture.
<code>styles</code>	An array of references to the style objects you want to use as overriding styles for the picture items. You may provide <code>nil</code> for this parameter if you do not want any overriding styles.
<code>inks</code>	An array of references to the ink objects you want to use as overriding inks for the picture items. You may provide <code>nil</code> for this parameter if you do not want any overriding inks.

## Picture Shapes

`transforms`

An array of references to the transform objects you want to use as overriding transforms for the picture items. You may provide `nil` for this parameter if you do not want any overriding transforms.

*function result* A reference to the newly created picture shape.

## DESCRIPTION

The `GXNewPicture` function creates a new picture shape.

In the `count` parameter, you specify the number of shapes you want to include as items of the picture, and in the `shapes` parameter, you provide references to the shapes.

In the `styles` parameter, you specify references to overriding styles. Each item of this array overrides the style of the corresponding shape in the `shapes` array. For example, the first style you provide in the `styles` array becomes the overriding style for the first shape in the `shapes` array, and so on. Similarly, in the `inks` and `transforms` parameters you specify references to overriding inks and transforms.

You may specify 0 for the `count` parameter and `nil` for the `shapes`, `styles`, `inks`, and `transforms` parameters to create an empty picture—a picture containing no picture items. You may provide `nil` for the `styles`, `inks`, or `transforms` parameters even if you provide shape references in the `shapes` parameter. In this case, the newly created picture shape contains picture items, but those items contain no overriding styles, inks, or transforms.

You may also provide `nil` for an individual item of a `styles`, `inks`, or `transforms` array if you do not want the corresponding picture item to have an overriding style, ink, or transform.

## SPECIAL CONSIDERATIONS

If no error results, the `GXNewPicture` function creates a picture shape; you are responsible for disposing of this shape when you no longer need it. See *Inside Macintosh: QuickDraw GX Objects* for information about creating and disposing of shapes.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

`out_of_memory`  
`parameter_is_nil`  
`shape_is_nil`  
`parameter_out_of_range`

**SEE ALSO**

For information about picture items and their overriding styles, inks, and transforms, see “About Picture Shapes” beginning on page 6-3.

For an example using this function, see “Creating and Drawing Picture Shapes” beginning on page 6-27.

To draw a picture shape once you’ve created one, use the `GXDrawShape` function, described in the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects*.

For information about disposing of picture shapes, see the description of the `GXDisposeShape` function, which is in the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects*.

## Getting and Setting Picture Geometries

---

This section describes the functions you can use to examine or replace the entire geometry of a picture shape—that is, all of the picture items included in the picture.

The `GXGetPicture` function provides references to the shapes contained in a picture geometry and references to their overriding styles, inks, and transforms.

The `GXSetPicture` function replaces references to the shapes contained in a picture geometry and references to their overriding styles, inks, and transforms.

## GXGetPicture

---

You can use the `GXGetPicture` function to obtain references to the shapes contained in a picture and references to their overriding styles, inks, and transforms.

```
long GXGetPicture(gxShape source, gxShape shapes[],
                 gxStyle styles[], gxInk inks[],
                 gxTransform transforms[]);
```

<code>source</code>	A reference to the picture shape whose items you want to examine.
<code>shapes</code>	An array of shape references. On return, this array contains references to the shapes contained in the source picture.
<code>styles</code>	An array of references to style objects. On return, this array contains references to the overriding styles contained in the source picture.
<code>inks</code>	An array of references to ink objects. On return, this array contains references to the overriding inks contained in the source picture.

## Picture Shapes

`transforms`

An array of references to transform objects. On return, this array contains references to the overriding transforms contained in the source picture.

*function result* The total number of items in the source picture.

## DESCRIPTION

If you provide arrays for the `shapes`, `styles`, `inks`, and `transforms` parameters, this function copies the references to shapes, styles, inks, and transforms from the picture's geometry into these arrays. However, you may provide `nil` for any of these parameters to indicate that you do not want to obtain the corresponding references.

Typically, you call this function twice. The first time you specify `nil` for all of the array parameters and use the function result to determine the number of picture items, which you can use to allocate arrays large enough to contain the shape, style, ink, and transform references. Then you call the function a second time to determine the actual references.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)
<code>cannot_set_item_shapes_to_nil</code>	(debugging version)
<code>cannot_use_original_item_shapes_when_growing_picture</code>	(debugging version)

**Warnings**

<code>index_out_of_range</code>	
<code>count_out_of_range</code>	
<code>picture_expected</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)
<code>picture_cannot_contain_itself</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)
<code>cannot_dispose_default_shape</code>	(debugging version)
<code>cannot_dispose_default_style</code>	(debugging version)
<code>cannot_dispose_default_ink</code>	(debugging version)
<code>cannot_dispose_default_transform</code>	(debugging version)
<code>cannot_dispose_default_colorProfile</code>	(debugging version)

**SEE ALSO**

For information about picture items and their overriding styles, inks, and transforms, see “About Picture Shapes” beginning on page 6-3.

For an example using this function, see “Getting and Setting Picture Geometries” beginning on page 6-31.

To examine a subset of the items in a picture geometry, use the `GXGetPictureParts` function, which is described on page 6-63.

To replace the information in the geometry of a picture shape, use the `GXSetPicture` function, which is described in the next section.

## **GXSetPicture**

---

You can use the `GXSetPicture` function to replace the information in the geometry of a picture shape.

```
void GXSetPicture(gxShape target, long count,
                  const gxShape shapes[], const gxStyle styles[],
                  const gxInk inks[],
                  const gxTransform transforms[]);
```

<code>target</code>	A reference to the picture shape whose geometry you want to replace.
<code>count</code>	The number of picture items in the new picture geometry.
<code>shapes</code>	An array of references to the shapes to include in the new picture geometry.
<code>styles</code>	An array of references to the styles you want to use as overriding styles in the new picture geometry. You may provide <code>nil</code> for this parameter if you do not want to change the existing overriding styles. You may provide the <code>gxSetToNil</code> constant to remove all of the existing overriding styles.
<code>inks</code>	An array of references to the inks you want to use as overriding inks in the new picture geometry. You may provide <code>nil</code> for this parameter if you do not want to change the existing overriding inks. You may provide the <code>gxSetToNil</code> constant to remove all of the existing overriding inks.
<code>transforms</code>	An array of references to the transforms you want to use as overriding transforms in the new picture geometry. You may provide <code>nil</code> for this parameter if you do not want to change the existing overriding transforms. You may provide the <code>gxSetToNil</code> constant to remove all of the existing overriding transforms.

## Picture Shapes

## DESCRIPTION

The `GXSetPicture` function replaces the geometry of the picture shape object referenced by the `target` parameter with a new geometry. To maintain correct owner counts, this function disposes of the shapes, styles, inks, and transforms referenced by the items of the original picture geometry.

In the `count` parameter, you specify the number of shapes in the new picture geometry, and in the `shapes` parameter you provide references to the shapes.

In the `styles` parameter, you specify references to the styles to use as overriding styles in the new picture geometry. Each item of this array overrides the style of the corresponding shape in the `shapes` array. For example, the first style you provide in the `styles` array becomes the overriding style for the first shape in the `shapes` array, and so on. Similarly, in the `inks` and `transforms` parameters you specify references to overriding inks and transforms.

You may specify 0 for the `count` parameter and `nil` for the `shapes`, `styles`, `inks`, and `transforms` parameters to create an empty picture—a picture containing no picture items. You may provide the `gxSetToNil` constant for the `styles`, `inks`, or `transforms` parameters even if you provide shape references in the `shapes` parameter. In this case, the newly created picture shape contains picture items, but those items contain no overriding styles, inks, or transforms, respectively.

You may also provide `nil` for an individual item of a `styles`, `inks`, or `transforms` array if you do not want the corresponding picture item to have an overriding style, ink, or transform.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_out_of_range</code>	(debugging version)
<code>cannot_set_item_shapes_to_nil</code>	(debugging version)
<code>cannot_use_original_item_shapes_when_growing_picture</code>	(debugging version)

**Warnings**

<code>picture_expected</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)
<code>picture_cannot_contain_itself</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)
<code>cannot_dispose_default_shape</code>	(debugging version)
<code>cannot_dispose_default_style</code>	(debugging version)
<code>cannot_dispose_default_ink</code>	(debugging version)
<code>cannot_dispose_default_transform</code>	(debugging version)
<code>cannot_dispose_default_colorProfile</code>	(debugging version)



**SEE ALSO**

For information about picture items and their overriding styles, inks, and transforms, see “About Picture Shapes” beginning on page 6-3.

To examine the items of a picture geometry, use the `GXGetPicture` function, which is described on page 6-59.

To replace a subset of the items in a picture geometry, use the `GXSetPictureParts` function, which is described on page 6-65.

For information about disposing of shapes, see the chapter “Shape Objects” of *Inside Macintosh: QuickDraw GX Objects*.

**Editing Picture Parts**

---

This section describes the functions you can use to examine and replace specific items within a picture geometry.

The `GXGetPictureParts` function allows you to obtain information about a specified subset of the picture items contained in a picture geometry.

The `GXSetPictureParts` function allows you to replace a subset of the picture items in a picture geometry with new picture items.

**GXGetPictureParts**

---

You can use the `GXGetPictureParts` function to obtain information about a specified subset of a picture’s items.

```
long GXGetPictureParts(gxShape source, long index, long count,
                      gxShape shapes[], gxStyle styles[],
                      gxInk inks[], gxTransform transforms[])
```

<code>source</code>	A reference to the picture shape whose items you want to examine.
<code>index</code>	The number of the first picture item you want to examine.
<code>count</code>	The total number of items you want to examine. You may supply the <code>gxSelectToEnd</code> constant (-1) to indicate that you want to examine all picture items (starting with the picture item indicated by the <code>index</code> parameter.)
<code>shapes</code>	An array of shape references. On return, this array contains references to the specified shapes contained in the source picture.
<code>styles</code>	An array of style references. On return, this array contains references to the overriding styles corresponding to the returned shapes.
<code>inks</code>	An array of ink references. On return, this array contains references to the overriding inks corresponding to the returned shapes.

## Picture Shapes

`transforms`

An array of transform references. On return, this array contains references to the overriding transforms corresponding to the returned shapes.

*function result* The total number of items returned.

## DESCRIPTION

The `GXGetPictureParts` function extracts information from a subset of the picture items in the picture shape referenced by the `source` parameter. You specify which picture items using the `index` and `count` parameters. The `index` parameter, which must have a value of 1 or greater, indicates the first picture item you want to examine. The `count` parameter indicates how many items you want to examine.

You provide arrays to hold the returned information in the `shapes`, `styles`, `inks`, and `transforms` parameters. In the `shapes` array, the `GXGetPictureParts` function returns references to the shapes that correspond to the picture items you specified with the `index` and `count` parameters. In the `styles`, `inks`, and `transforms` arrays, this function returns references to the overriding styles, inks, and transforms for the specified picture items. You may provide `nil` for any of the array parameters to indicate that you do not want to obtain the corresponding references.

This function returns as its function result the number of picture items returned. Typically, this value is the same as the value you provide for the `count` parameter.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)

**Warnings**

<code>index_out_of_range</code>	
<code>count_out_of_range</code>	
<code>picture_expected</code>	(debugging version)

## SEE ALSO

For information about picture items and their overriding styles, inks, and transforms, see “About Picture Shapes” beginning on page 6-3.

## Picture Shapes

To examine all of the items in a picture geometry, use the `GXGetPicture` function, which is described on page 6-59.

To replace a subset of the items in a picture geometry, use the `GXSetPictureParts` function, which is described in the next section.

## GXSetPictureParts

---

You can use the `GXSetPictureParts` function to add, remove, or replace a range of picture items in a picture shape's geometry.

```
void GXSetPictureParts(gxShape target, long index, long oldCount,
                      long newCount, const gxShape shapes[],
                      const gxStyle styles[],
                      const gxInk inks[],
                      const gxTransform transforms[]);
```

<code>target</code>	A reference to the picture shape whose picture item list you want to alter.
<code>index</code>	The number of the first picture item you want to replace.
<code>oldCount</code>	The total number of picture items you want to replace. A value of 0 indicates that you want to insert new picture items before the existing picture item indicated by the <code>index</code> parameter, rather than replace items. You may supply the <code>gxSelectToEnd</code> constant (-1) to indicate that you want to replace all picture items (starting with the picture item indicated by the <code>index</code> parameter.)
<code>newCount</code>	The total number of new picture items to insert in the picture. A value of 0 specifies that you do not want to insert new items into the picture; instead, the existing items you specified with the <code>index</code> and <code>oldCount</code> parameters are removed.
<code>shapes</code>	An array of references to the shapes to include as the new picture items in the new picture geometry.
<code>styles</code>	An array of references to the style objects you want to use as overriding styles in the new picture geometry. You may provide <code>gxSetToNil</code> for this parameter if you do not want any overriding styles.
<code>inks</code>	An array of references to the ink objects you want to use as overriding inks in the new picture geometry. You may provide <code>gxSetToNil</code> for this parameter if you do not want any overriding inks.
<code>transforms</code>	An array of references to the transform objects you want to use as overriding transforms in the new picture geometry. You may provide <code>gxSetToNil</code> for this parameter if you do not want any overriding transforms.

## Picture Shapes

## DESCRIPTION

The `GXSetPictureParts` function allows you to insert new picture items in a picture, to remove picture items from a picture, or to replace picture items with new picture items. In any of these three cases, the `target` parameter specifies the picture to be modified, the `oldCount` parameter specifies the number of items to remove, the `newCount` parameter specifies the number of items to add, and the `shapes`, `styles`, `inks`, and `transforms` parameters specify the information for the new picture items.

- `n` To insert picture items, set the `oldCount` parameter to 0. Use the `index` parameters to specify where to add the new picture items. (This function inserts the new picture items before the existing item you specify with the `index` parameter. For example, if you specify 1 for this parameter, the new picture items are inserted before the first item of the existing picture item list.)
- `n` To remove picture items, set the `newCount` parameter to 0 and the `shapes`, `styles`, `inks`, and `transforms` parameters to `nil`. Use the `index` and `oldCount` parameters to specify which picture items to remove.
- `n` To replace picture items, use the `index` and `oldCount` parameters to specify the existing picture items to remove and use the `newCount`, `shapes`, `styles`, `inks`, and `transforms` parameters to specify the new picture items to insert in their place.

To maintain correct owner counts, this function clones the inserted shapes, styles, inks and transforms, and disposes of any replaced shapes, styles, inks, and transforms.

## ERRORS, WARNINGS, AND NOTICES

## Errors

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)
<code>cannot_set_item_shapes_to_nil</code>	(debugging version)
<code>cannot_use_original_item_shapes_when_growing_picture</code>	(debugging version)

## Warnings

<code>index_out_of_range</code>	
<code>count_out_of_range</code>	
<code>picture_expected</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)
<code>picture_cannot_contain_itself</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)
<code>cannot_dispose_default_shape</code>	(debugging version)
<code>cannot_dispose_default_style</code>	(debugging version)
<code>cannot_dispose_default_ink</code>	(debugging version)
<code>cannot_dispose_default_transform</code>	(debugging version)
<code>cannot_dispose_default_colorProfile</code>	(debugging version)

**SEE ALSO**

For information about picture items and their overriding styles, inks, and transforms, see “About Picture Shapes” beginning on page 6-3.

For examples using this function, see “Removing and Replacing Items in a Picture” beginning on page 6-35.

To extract information from a subset of the items contained in a picture shape’s geometry, use the `GXGetPictureParts` function, which is described on page 6-63.

To replace every item in a picture geometry, use the `GXSetPicture` function, which is described on page 6-61.

For information about disposing of shapes, see the chapter “Shape Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## Drawing Pictures

---

QuickDraw GX provides two methods of drawing a picture:

- n You can create a picture shape (by calling the `GXNewPicture` function, by copying an existing picture shape, and so on) and use the `GXDrawShape` function to draw the picture.
- n You can create an array of shape references, and arrays of references to overriding styles, inks, and transforms, and use the `GXDrawPicture` function to draw the corresponding picture.

In general, you should use the `GXDrawShape` function to draw any QuickDraw GX graphic, including picture shapes. In fact, the `GXDrawPicture` function creates a temporary picture shape, uses the `GXDrawShape` function to draw it, and then disposes of it. The `GXDrawShape` function is described in the “Shape Objects” chapter of *Inside Macintosh: QuickDraw GX Objects*.

You would typically use the `GXDrawPicture` function only in simple situations—for example, if you knew you wanted to draw a particular picture only once.

## GXDrawPicture

---

You can use the `GXDrawPicture` function to draw a picture without encapsulating the items of the picture geometry in a picture shape.

```
void gxDrawPicture(long count, const gxShape shapes[],
                  const gxStyle styles[], const gxInk inks[],
                  const gxTransform transforms[]);
```

<code>count</code>	The number of picture items in the new picture shape.
<code>shapes</code>	An array of references to the shapes you want to draw.

## Picture Shapes

<code>styles</code>	An array of references to the style objects you want to use to override the styles of the shapes specified in the <code>shapes</code> parameter. You may provide <code>nil</code> for this parameter if you do not want any overriding styles.
<code>inks</code>	An array of references to the ink objects you want to use to override the inks of the shapes specified in the <code>shapes</code> parameter. You may provide <code>nil</code> for this parameter if you do not want any overriding inks.
<code>transforms</code>	An array of references to the transform objects you want to use to override the transforms of the shapes specified in the <code>shapes</code> parameter. You may provide <code>nil</code> for this parameter if you do not want any overriding transforms.

## DESCRIPTION

The `GXDrawPicture` function allows you to draw a picture without having to create a picture shape yourself. Instead, you specify the items of a picture geometry using the `shapes`, `styles`, `inks`, and `transforms` parameters.

The `GXDrawPicture` function creates a temporary picture shape using the values specified in these arrays, and draws the picture shape using the `GXDrawShape` function. The `GXDrawPicture` function calls the `GXNewPicture` function to create the temporary picture shape.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

<code>out_of_memory</code>	
<code>shape_is_nil</code>	
<code>inconsistent_parameters</code>	(debugging version)
<code>parameter_out_of_range</code>	(debugging version)
<code>index_is_less_than_one</code>	(debugging version)
<code>count_is_less_than_one</code>	(debugging version)
<code>cannot_set_item_shapes_to_nil</code>	(debugging version)
<code>cannot_use_original_item_shapes_when_growing_picture</code>	(debugging version)

**Warnings**

<code>index_out_of_range</code>	
<code>count_out_of_range</code>	
<code>picture_expected</code>	(debugging version)
<code>shape_access_not_allowed</code>	(debugging version)
<code>picture_cannot_contain_itself</code>	(debugging version)
<code>cannot_dispose_locked_tag</code>	(debugging version)
<code>cannot_dispose_default_shape</code>	(debugging version)
<code>cannot_dispose_default_style</code>	(debugging version)
<code>cannot_dispose_default_ink</code>	(debugging version)
<code>cannot_dispose_default_transform</code>	(debugging version)
<code>cannot_dispose_default_colorProfile</code>	(debugging version)

**SEE ALSO**

For information about picture items and their overriding styles, inks, and transforms, see “About Picture Shapes” beginning on page 6-3.

For an example using this function, see “Creating and Drawing Picture Shapes” beginning on page 6-27.

To encapsulate a picture geometry in a picture shape, use the `GXNewPicture` function, described on page 6-57.

To draw a picture shape, use the `DrawShape` function, described in the “Shape Objects” chapter of *Inside Macintosh: QuickDraw GX Objects*.

## Hit-Testing Pictures

---

This section describes the `GXHitTestPicture` function. To hit-test a picture, this function

- n hit-tests each shape contained in the picture
- n compiles a list of shapes that were hit
- n selects one of the shapes using criteria you provide
- n provides information about the shape in the picture that was hit

For more information about how QuickDraw GX hit-tests shapes, see the chapter “Shape Objects” and the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## GXHitTestPicture

---

You can use the `GXHitTestPicture` function to determine whether a test point hits a picture shape and to discover which shape in the picture hierarchy is hit.

```
gxShape GXHitTestPicture(gxShape target, const gxPoint *test,
                        gxHitTestInfo *result, long level,
                        long depth);
```

<code>target</code>	A reference to the picture shape to hit-test.
<code>test</code>	A pointer to a <code>gxPoint</code> structure. The <code>GXHitTestPicture</code> function determines whether the location specified by this point hits the target picture.
<code>result</code>	A pointer to a <code>gxHitTestInfo</code> structure. On return, this structure contains information identifying the part of the target picture that was hit by the test point.

## Picture Shapes

<code>level</code>	A level in the picture hierarchy. This parameter, along with the <code>depth</code> parameter, is used to determine which shape in the picture to return as the function result. You must provide a nonnegative value for this parameter. A value of 0 indicates you want the item at the lowest level of the hierarchy.
<code>depth</code>	A shape depth in the picture as drawn. This parameter, along with the <code>level</code> parameter, is used to determine which shape in the picture to return as the function result. You must provide a nonnegative value for this parameter. A value of 0 indicates you want the hit item at the lowest depth.
<i>function result</i>	A reference to the shape (at the specified shape depth and hierarchy level) hit by the test point. The function result is <code>nil</code> if no shape was hit that satisfies the criteria.

## DESCRIPTION

The `GXHitTestPicture` function compares the point indicated by the `test` parameter with each shape in the picture referenced by the `target` parameter. To determine whether the test point hits a shape, this function uses the hit-test parameters contained in that shape's transform object, or contained in the overriding transform if there is one.

If the target picture contains shapes that overlap when drawn, more than one shape might be hit by the test point. The function uses the `depth` parameter to select which of these shapes is the hit shape. If you set this parameter to 1, the function selects the frontmost shape as the hit shape. If you set this parameter to 2, the function selects the shape immediately behind the frontmost shape as the hit shape, and so on.

Before returning a reference to the hit shape, this function examines how deep into the target picture's hierarchy the hit shape is. If the hit shape is deeper into the hierarchy than the level indicated by the `level` parameter, this function does not return a reference to the hit shape. Instead, it returns a reference to the subpicture at the appropriate level of the target picture's hierarchy that contains the hit shape.

For example, if the hit shape is at level 2 of the picture hierarchy—that is, it is an item of a picture which is an item of the target picture—then specifying a value of 2 for the `level` parameter causes the function to return a reference to the shape as the function result. However, if you specify a value of 1 for the `level` parameter, the function returns a reference to the picture that contains the hit shape, rather than a reference to the hit shape itself. Specifying a level of 0 indicates you want the item at the lowest level of the picture hierarchy.



## Picture Shapes

This function also returns information in the `gxHitTestInfo` structure pointed to by the `result` parameter:

- n The `what` field indicates which shape part of the hit shape was hit by the test point.
- n The `index` field indicates the index of the geometric point hit by the test point.
- n The `distance` field indicates the distance of the test point from the shape part hit.
- n The `which` field contains a reference to the hit shape.
- n The `containerPicture` field contains a reference to the picture that contains the hit shape.
- n The `containerIndex` field indicates the index of the hit shape within the container picture.
- n The `totalIndex` field indicates the overall index of the hit shape within the target picture.

For more information about the `gxHitTestInfo` structure, see the chapter “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

## ERRORS, WARNINGS, AND NOTICES

**Errors**

`out_of_memory`  
`shape_is_nil`  
`parameter_is_nil`  
`parameter_out_of_range` (debugging version)  
`parameter_out_of_range`

**Warnings**

`character_substitution_took_place`  
`font_substitution_took_place`  
`picture_expected` (debugging version)  
`unable_to_traverse_open_contour_that_starts_or_ends_off_the_curve` (debugging version)

## SEE ALSO

For more information about hit-testing shapes, see the chapters “Shape Objects” and “Transform Objects” in *Inside Macintosh: QuickDraw GX Objects*.

For examples using this function, see “About Hit-Testing Picture Shapes” beginning on page 6-24.

## Summary of Picture Shapes

---

### Functions

---

#### Creating Picture Shapes

```
gxShape GXNewPicture      (long count, const gxShape shapes[],
                           const gxStyle styles[], const gxInk inks[],
                           const gxTransform transforms[])
```

#### Getting and Setting Picture Geometries

```
long GXGetPicture      (gxShape source, gxShape shapes[],
                       gxStyle styles[], gxInk inks[],
                       gxTransform transforms[]);

void GXSetPicture      (gxShape target, long count,
                       const gxShape shapes[], const gxStyle styles[],
                       const gxInk inks[],
                       const gxTransform transforms[]);
```

#### Editing Picture Parts

```
long GXGetPictureParts  (gxShape source, long index, long count,
                       gxShape shapes[], gxStyle styles[],
                       gxInk inks[], gxTransform transforms[])

void GXSetPictureParts  (gxShape target, long index, long oldCount,
                       long newCount, const gxShape shapes[],
                       const gxStyle styles[],
                       const gxInk inks[],
                       const gxTransform transforms[]);
```

#### Drawing Pictures

```
void gxDrawPicture      (long count, const gxShape shapes[],
                       const gxStyle styles[], const gxInk inks[],
                       const gxTransform transforms[]);
```

#### Hit-Testing Pictures

```
gxShape GXHitTestPicture (gxShape target, const gxPoint *test,
                       gxHitTestInfo *result, long level,
                       long depth);
```

# Glossary

---

**attributes** A property of many QuickDraw GX objects. The attributes property of an object is a set of flags that control various aspects of that object's behavior.

**bitmap** A QuickDraw GX data structure that describes a pixel map on a physical device. A bitmap structure is a property of a view device object.

**bitmap color profile** The object that specifies color-matching information about the device on which a bitmap was created.

**bitmap color set** An array of color values associated with a bitmap. If a bitmap uses a color set (as opposed to a color space), each pixel value in the bitmap's pixel image represents an index into this color set.

**bitmap color space** A color space associated with a bitmap. If a bitmap uses a color space (as opposed to a color set), each pixel value in the bitmap's pixel image represents a color value in this color space.

**bitmap height** The number of pixels in each column of a bitmap.

**bitmap position** The position of the upper-left corner of a bitmap in geometry space.

**bitmap shape** A type of QuickDraw GX shape. The geometry of a bitmap shape contains a pixel image and color information.

**bitmap width** The number of pixels in each row of a bitmap.

**bounding rectangle** The smallest rectangle that encloses a shape. The coordinates of a bounding rectangle are ordered.

**bytes per row** The number of bytes in a pixel image required to represent each row of a bitmap.

**cap** See **cap property**.

**cap attributes** A set of flags that modify the way QuickDraw GX draws cap shapes.

**cap property** A property of a style object that is used to specify how the end points of contours are drawn.

**cap shape** A shape drawn at the end points of another shape's contours.

**color ramp** A shape that blends from one color to another.

**contour** A connected series of lines and curves. The geometry property of a geometric shape is made up of one or more contours.

**contour direction** A value, either clockwise or counterclockwise, that QuickDraw GX assigns to each contour in a shape's geometry.

**contour index** A number used to specify a particular geometric point in a contour: the first geometric point in a contour has contour index 1, the second has contour index 2, and so on. See also **geometry index**.

**control bits** A set of bit flags in a path geometry that determines which geometric points are on curve and which are off curve.

**control point** A geometric point used to control the curvature of a curve.

**curve error** A property of the style object used to specify the accuracy of certain operations, such as converting paths to polygons.

**curve join** A join attribute specifying that a shape should be drawn with curved corners.

**curve shape** A type of QuickDraw GX shape. The geometry of a curve shape defines a Bézier curve.

**dash** (n) See **dash property**. (v) Applying a dash shape to the contours of another shape.

**dash advance** The distance between dashes in a dashed contour.

**dash attributes** A set of flags that modify the way QuickDraw GX dashes a shape.

**dash phase** How far into a dash a contour begins.

**dash property** A property of the style object used to draw contours as repeated patterns of shapes rather than continuous lines.

**dash scale** The factor to divide by when scaling a dash shape perpendicularly to the dashed shape's contours.

**dash shape** A shape used to dash the contours of another shape.

**dashed shape** A shape whose contours have been drawn with a dash shape.

**depth** A number indicating the position in front to back order at which a picture item is drawn. The greater a shape's depth, the more other shapes are drawn on top of the shape.

**disk-based pixel image** A bitmap pixel image that is stored in a file (rather than in memory) even when the bitmap shape is memory.

**dither** To approximate colors that a display device cannot draw with patterns of similar colors that the display device can draw.

**empty shape** A type of QuickDraw GX shape. Empty shapes have no geometry, are contained by every other shape, and do not appear when drawn.

**edge** A line or curve that makes up part of a shape contour.

**even-odd rule** A rule used when drawing filled shapes to determine which areas are filled. The even-odd rule does not fill areas which lie under overlapping contours. Compare **winding-number rule**.

**fill** See **shape fill**.

**framed fill** A shape fill that indicates a shape's geometry describes an outline—the outline defined by the contours of the shape's geometry. Framed fills include open-frame fill and closed-frame fill.

**framed shape** A shape that describes an outline—the outline defined by the contours of the shape's geometry. The shape fill of a framed shape can be open-frame fill or closed-frame fill.

**full shape** A type of QuickDraw GX shape. Full shapes have no geometry, contain every other shape, and cover all area when drawn.

**geometry index** A number used to specify a particular geometric point in a geometry: the first geometric point in a geometry has geometry index 1, and so on. Whereas contour indices start over with each contour in a geometry, geometry indices do not.

**geometric pen** The pen used by QuickDraw GX to draw framed shapes. The width and placement of this pen are affected by style properties.

**geometric point** An (x, y) coordinate pair used to specify a location in a shape's geometry. Geometric points can specify the ends of lines or curves or the off-curve control points used to control curvature.

**geometric shape** Any QuickDraw GX shape that has one of the following shape types: empty, full, point, line, curve, rectangle, polygon, path.

**geometry** A property of a QuickDraw GX shape object. A shape's geometry is the specification of the actual size, position, and form of the shape. For example, for a line shape, the geometry specifies the locations (in local coordinates) of the end points of the line.

**grid point** (1) A location in the QuickDraw GX coordinate system. Grid points are infinitely thin, and fall between pixels. (2) The distance between two grid points.

**halftone** A QuickDraw GX data structure that specifies a pattern and a set of colors. A halftone is used to achieve a greater range of colors than may be available on a display device.

**index** A number that indicates the position of and item in a list. See also **contour index** and **geometric index**.

**inverse fill** A shape fill that indicates a shape's geometry describes an area—the area not contained within the contours of the shape's geometry. Inverse fills include inverse even-odd fill, and inverse winding fill.

**join** See **join property**.

**join attributes** A set of flags that modify the way QuickDraw GX adds a join shape to the corners of a shape.

**join property** A property of a style object that specifies how the corners of a geometric shape should be drawn.

**join shape** A shape drawn at the corners of another shape.

**layout shape** A type of QuickDraw GX shape. The geometry of a layout shape contains a line of text and sophisticated typographic formatting information.

**level** A number indicating how many pictures separate a shape from the root picture in a picture hierarchy.

**level cap** A cap shape that is not rotated to match the angle of the contour on which it is drawn.

**level join** A join shape that is not rotated to match the angle that bisects the corner on which it is drawn.

**miter** The length a sharp join can reach before being truncated.

**off-curve control point** See **control point**.

**offscreen bitmap** A bitmap that exists in memory or on disk but is not associated with a physical display device.

**overriding ink object** An optional part of a picture item. If a picture item has an overriding ink object, QuickDraw GX uses the information in the overriding ink when drawing the item, rather than the information in the original ink object.

**overriding style object** An optional part of a picture item. If a picture item has an overriding style object, QuickDraw GX uses the information in the overriding style when drawing the item, rather than the information in the original style object.

**overriding transform object** An optional part of a picture item. If a picture item has an overriding transform object, QuickDraw GX uses the information in the overriding transform when drawing the item, rather than the information in the original transform object.

**path contour** A connected series of straight lines and curves.

**path shape** A type of QuickDraw GX shape. The geometry of a path shape is made up of zero, one, or more path contours.

**pattern** See **pattern property**.

**pattern grid** A pair of vectors that determine the placement of a pattern shape over the area of another shape.

**pattern property** A property of a style object that specifies how the area of a shape is to be filled.

**pattern shape** A shape copied over the area of another shape at positions specified by a pattern grid.

**picture hierarchy** A picture shape that contains other picture shapes as items.

**picture item** An element of a picture shape's geometry. Each picture item contains a reference to a shape and, optionally, a reference to an overriding style, an overriding ink, and an overriding transform.

**picture shape** A type of QuickDraw GX shape that represents a collection of other shapes.

**pixel depth** See **pixel size**.

**pixel image** A two-dimensional array of pixel values, each of which describes the color of one pixel in a bitmap.

**pixel size** The number of bits required to represent the color information for each pixel in a bitmap. Also called *pixel depth*.

**pixel value** A series of bits in a bitmap's pixel image that represent a single pixel of the bitmap. This value can represent a color value (if the bitmap uses a color space) or an index into a color set (if the bitmap uses a color set).

**point** See **control point**, **geometric point**, **grid point**, and **point shape**.

**point shape** A type of QuickDraw GX shape. The geometry of a point shape specifies an x-coordinate and a y-coordinate. Point shapes appear as a single pixel (if the pen width is 0) or as a cap shape (if the pen width is greater than 0).

**polygon contour** A connected series of straight lines.

**polygon shape** A type of QuickDraw GX shape. The geometry of a polygon shape is made up of zero, one, or more polygon contours.

**primitive form** Having stylistic variations incorporated into the shape type, geometry, and shape fill.

**primitive shape** A shape whose shape type, geometry, and shape fill have had stylistic information incorporated into them.

**property** An item or set of data in a QuickDraw GX object. A property of an object is analogous to a field of a data structure; however, a field is accessed through its name, whereas a property is accessed through a function.

**rectangle shape** A type of QuickDraw GX shape. The geometry of a rectangle shape contains points representing two opposing corners of a rectangle.

**reduce** To remove unnecessary geometric points from a geometry.

**RGB color space** A color space whose three components measure the intensity of red, green, and blue. Used mostly for color video.

**shape** (1) A graphic or typographic item (such as a geometric shape, a bitmap, or a line of text) created and drawn with QuickDraw GX. (2) A set of QuickDraw GX objects that, taken together, describe the type and characteristics of such a graphic or typographic item. A shape consists of a shape object, a style object, an ink object, and a transform object.

**shape fill** A property of a shape object. The shape fill specifies whether and how QuickDraw GX fills in the outlines of a shape that it draws.

**shape type** A property of a shape object. The shape type specifies the classification (such as point, line, bitmap, or text) of a particular shape.

**sharp join** A join attribute specifying that a shape should be drawn with sharp corners.

**simplify** To remove crossed and overlapping contours from a geometry.

**solid fill** A shape fill that indicates a shape's geometry describes an area—the area surrounded by the contours of the shape's geometry. Solid fills include even-odd fill, winding fill, inverse even-odd fill, and inverse winding fill.

**solid shape** A shape that describes an area—the area surrounded by the contours of the shape's geometry. The shape fill of a solid shape can be even-odd shape fill, winding-number shape fill, or one of the inverse shape fills.

**standard cap** A type of cap. Standard caps are square caps and semicircular caps.

**standard join** A type of join. Standard joins are sharp joins and curve joins.

**style** See **style object**.

**style attributes** A property of a style object. Style attributes are a set of flags that influence how the information in a style object affects a shape.

**style object** A QuickDraw GX object associated with a shape object. A style object contains information that affects the visual appearance of a shape when it is drawn.

**style property** One of the pieces of information stored in a style object and maintained by QuickDraw GX.

**style reference** A reference to a style object.

**text shape** A type of QuickDraw GX shape. The geometry of a text shape contains a string of characters to be drawn in a single font and style.

**transform concatenation** The process by which QuickDraw GX combines the clips and mappings of transform objects at different levels of a picture hierarchy when drawing a picture shape.

**true inside** The right side of a clockwise contour or the left side of a counterclockwise contour.

**type** See **shape type**.

## G L O S S A R Y

**type conversion** The process of changing a shape from one shape type to another. Often the geometry of the shape is significantly affected during this process.

**typographic shape** Any QuickDraw GX shape that has one of the following shape types: text, glyph, layout.

**unique items attribute** A shape attribute that affects the way items are added to picture shapes.

**winding-number rule** A rule used when drawing filled shapes to determine which areas are filled. The winding-number rule fills areas that lie under overlapping contours. Compare **even-odd rule**.





# Index

---

## A

---

area of a shape 4-17, 4-45 to 4-47, 4-88  
arithmetic operations on shapes. *See* geometric arithmetic  
attributes 3-102  
auto-advance dash attribute 3-70 to 3-72, 3-105  
auto-inset style attribute 3-19 to 3-20, 3-99

---

## B

---

bend dash attribute 3-74 to 3-79, 3-105  
Bézier curves 2-18 to 2-20  
bitmap color profiles 5-5  
bitmap color sets 5-5  
bitmap color spaces 5-5  
bitmap data source alias structure 5-65  
bitmap geometries  
    editing 5-53 to 5-54, 5-71 to 5-76  
    fields of 5-4 to 5-7  
    replacing 5-68 to 5-71  
    structure of 5-62 to 5-63  
bitmap geometry structure 5-62  
bitmap height 5-5 to 5-7  
bitmap shapes 5-3 to 5-80  
    and view devices 5-45 to 5-51  
    applying transfer modes to 5-9, 5-32 to 5-34  
    black-and-white 5-15 to 5-21  
    clipping 5-43  
    color 5-21 to 5-29  
    converting other shapes to 5-34 to 5-38  
    creating and drawing 5-15 to 5-29, 5-66, 5-77  
    dithering 5-30 to 5-32  
    drawing with halftones 5-30 to 5-32  
    functions for  
        functions specific to bitmaps 5-65 to 5-80  
        other applicable functions 5-54 to 5-61  
    introduced 1-17 to 1-19  
    mapping 5-10, 5-38 to 5-43  
    and map-transform shape attribute 5-40  
    offscreen 5-14, 5-45 to 5-52  
    rotating 5-40  
    scaling 5-41 to 5-43  
    skewing 5-39  
    and view devices 5-12, 5-14  
bitmap width 5-5 to 5-7  
black-and-white bitmaps 5-15 to 5-21

bounding rectangle of a shape  
    determining 4-43, 4-90  
    setting 4-47 to 4-49, 4-92  
break dash attribute 3-74 to 3-79, 3-105  
bytes per row in bitmap geometries 5-5 to 5-7  
    unaligned 5-19

---

## C

---

cap attributes 3-24, 3-59, 3-100  
caps 3-23 to 3-24. *See also* cap style property  
    adding to a shape 3-57 to 3-61  
    definition of cap structure 3-99 to 3-100  
    functions for 3-123 to 3-129  
    interactions with joins, dashes, patterns 3-33 to 3-35, 3-91 to 3-95  
    level 3-24, 3-101  
    standard 3-24, 3-59 to 3-61  
cap structure 3-99 to 3-100  
cap style property 3-23, 3-25. *See also* caps  
    functions for 3-123 to 3-129  
center-frame style attribute 3-17, 3-98  
center of a shape 4-87  
clip dash attribute 3-29, 3-69, 3-105  
clipping  
    dashes 3-29, 3-69, 3-105  
clockwise contour direction 4-7  
closed-frame shape fill  
    and multiple contours 2-62, 2-64  
    compared to even-odd shape fill 2-21, 2-23  
    compared to open-frame shape fill 2-23  
    and crossed contours 2-51  
    defined 2-13  
    and overlapping contours 2-53  
colinear geometric points 4-30  
color bitmaps 5-21 to 5-29  
color profiles  
    of bitmap shapes 5-5  
color ramps 5-25  
color set objects  
    of bitmap shapes 5-5  
color spaces  
    of bitmap shapes 5-5  
concatenating transforms 6-19 to 6-23, 6-44 to 6-45  
containment, testing shapes for 4-18 to 4-21, 4-58 to 4-59, 4-100 to 4-104

- contour direction
  - defined 2-11, 4-7
  - determining 4-68
  - effect on shape fill 4-23 to 4-27
  - reversing 4-23 to 4-27, 4-70
- contour index 2-23
- contours 4-4 to 4-9. *See also* path contours; path shapes; polygon contours; polygon shapes
  - breaking 4-28 to 4-30, 4-72
  - counting 2-136, 4-30
  - crossed
    - creating 2-50
    - effect of shape fill on 2-14 to 2-15, 2-24
    - and pen placement 3-19, 3-56
    - removing 4-76
  - defined 2-9
  - determining direction 4-25, 4-68
  - finding a specific point on 4-42, 4-85
  - left side 4-5
  - overlapping
    - creating 2-52
    - effect of shape fill on 2-14 to 2-15, 2-26, 2-63 to 2-65
    - removing 4-76
  - removing unnecessary contour breaks 4-76
  - right side 4-5
  - true inside 4-9
- control bits of path geometries 2-25, 2-56
- control points 2-18, 2-25, 2-56
- coordinate spaces 3-20 to 3-21
- counterclockwise contour direction 4-7
- crossed contours
  - creating 2-50
  - effect of shape fill on 2-14 to 2-15, 2-24
  - and pen placement 3-19, 3-56
  - removing 4-33, 4-76
- curve error style property 3-14
  - effect when converting shapes 3-45 to 3-48
  - effect when reducing 3-49 to 3-51
  - functions for 3-114 to 3-118
- curve geometries
  - defined 2-18
  - determining 2-125
  - editing 2-79 to 2-81, 2-126
  - structure of 2-105
- curve join attribute 3-103
- curve joins 3-26, 3-103
- curve shapes
  - converting other shapes to 2-71 to 2-74
  - creating and drawing 2-41 to 2-42, 2-113, 2-159
  - default 2-20
  - defined 2-18
  - dividing in two 2-19

## D

---

- dash advance 3-29, 3-66 to 3-67, 3-104
- dash attributes
  - as field of dash record 3-104
  - auto-advance dash attribute 3-70 to 3-72, 3-105
  - bend dash attribute 3-74 to 3-79, 3-105
  - break dash attribute 3-74 to 3-79, 3-105
  - clip dash attribute 3-29, 3-105
  - defined 3-66
  - as enumeration 3-105, 3-106
  - level dash attribute 3-105
- dashes 3-27 to 3-31. *See also* dash style property
  - adding to a shape 3-66 to 3-70
  - adjusting to fit contours 3-70 to 3-74
  - auto-advancing 3-70 to 3-74, 3-105
  - bending 3-30, 3-74 to 3-79, 3-105
  - breaking 3-29, 3-74 to 3-79, 3-105
  - clipping 3-29, 3-69, 3-105
  - definition of dash structure 3-103 to 3-104
  - effect of shape fill 3-103
  - functions for 3-134 to 3-141
  - height of 3-28, 3-104
  - insetting 3-73 to 3-74, 3-99
  - interactions with caps, joins, patterns 3-33 to 3-35, 3-91 to 3-95
  - level 3-105
  - phasing 3-69, 3-104
  - positions, determining 3-81 to 3-85, 3-140
  - scaling 3-28, 3-68, 3-104
  - text used as 3-80 to 3-81
- dash phase 3-29, 3-104
- dash scale 3-28, 3-68, 3-104
- dash structures 3-103 to 3-104
- dash style property 3-27
  - functions for 3-134 to 3-141
- default shapes
  - curve 2-20
  - line 2-17
  - path 2-26
  - point 2-16
  - polygon 2-24
  - rectangle 2-21
- depth of shapes 6-24
- depth of picture items 6-51
- device-grid style attribute 3-21, 3-42 to 3-45, 3-99
- difference operation 4-21, 4-63, 4-110
- disk-based pixel images 5-44 to 5-45
- dithering, bitmaps 5-30 to 5-32
- disk-based pixel images 5-7
- duplicate geometric points 4-30

## E

---

empty shapes  
     creating and drawing 2-29  
     defined 2-16  
 end caps 3-57 to 3-61, 3-100. *See also* caps  
 even-odd rule for filling shapes 2-13  
 even-odd shape fill  
     compared to closed-frame shape fill 2-21, 2-23  
     compared to open-frame shape fill 2-23  
     compared to winding fill 2-13  
     compared to winding shape fill 2-24  
     and concentric contours 2-63, 2-64  
     and crossed contour 2-51  
     defined 2-13  
     and overlapping contour 2-53  
 exclusion operation 4-21, 4-64, 4-114

## F

---

framed shape fills 2-12. *See also* closed-frame shape fill;  
     open-frame shape fill  
 full shapes  
     creating and drawing 2-29  
     defined 2-16

## GA–GXA

---

geometric arithmetic  
     difference 4-63, 4-110  
     examples of 4-60 to 4-66  
     exclusion 4-64, 4-114  
     functions for 4-104 to 4-116  
     intersection 4-61, 4-105, 4-107  
     introduced 4-21 to 4-22  
     inversion 4-65, 4-116  
     reverse difference 4-64, 4-112  
     union 4-62, 4-106, 4-109  
 geometric information 4-41 to 4-47  
     functions for 4-83 to 4-92  
     introduced 4-16  
     shape area 4-45 to 4-47, 4-88  
     shape bounds 4-43, 4-47 to 4-49, 4-90, 4-92  
     shape center 4-87  
     shape length 4-42, 4-83  
     shape length to point 4-42, 4-85  
 geometric pen. *See* pen, geometric  
 geometric points  
     colinear 4-30  
     duplicate 4-30  
     effect of fractional coordinate values 2-40

    removing unnecessary 4-30  
     replacing 2-79 to 2-99, 2-142 to 2-157  
 geometric shapes 2-5 to 2-166. *See also* curve shapes;  
     empty shapes; full shapes; line shapes; path  
     shapes; point shapes; polygon shapes; rectangle  
     shapes  
     adding caps to 3-123 to 3-129  
     adding dashes to 3-134 to 3-141  
     adding joins to 3-129 to 3-134  
     adding patterns to 3-142 to 3-148  
     caps, adding 3-57 to 3-61  
     converting between types 2-65 to 2-78, 2-101 to 2-102  
     creating 2-28 to 2-65, 2-109 to 2-119  
     dashes, adding 3-66 to 3-81  
     data structures for 2-104 to 2-108  
     editing 2-79 to 2-99, 2-119 to 2-157  
     functions for  
         functions specific to geometric shapes 2-108 to  
             2-162  
         other applicable functions 2-100 to 2-101  
     introduced 1-7  
     joins, adding 3-61 to 3-66  
     patterns, adding 3-86 to 3-88  
     stylistic variations. *See* style properties of geometric  
     shapes  
 geometries  
     constraining to grids 3-40 to 3-45  
     editing 2-93 to 2-99, 2-135 to 2-157  
     incorporating style information into 4-38 to 4-40,  
         4-79  
     of point shapes 2-9  
     removing unnecessary points 4-10, 4-30, 4-74  
     replacing 2-119 to 2-134  
 geometry index 2-23  
 graphics pen. *See* pen, geometric  
 grids 3-20 to 3-21  
     constraining geometries to 3-40 to 3-42  
     for patterns 3-32, 3-107

## GXB

---

gxBitmapDataSourceAlias structure 5-65  
 gxBitmap structure 5-62  
 GXBreakShape function 4-28 to 4-30, 4-72

## GXC

---

gxCapAttributes enumeration 3-101  
 gxCapRecord structure 3-99  
 GXContainsBoundsShape function 4-58 to 4-59, 4-101  
 GXContainsRectangle function 4-58 to 4-59, 4-100

GXContainsShape **function** 4-58 to 4-59, 4-103  
 GXCountShapeContours **function** 2-136, 4-30  
 GXCountShapePoints **function** 2-137  
 gxCurve **structure** 2-105

## GXD

---

gxDashAttributes **enumeration** 3-105  
 gxDashRecord **structure** 3-103  
 GXDifferenceShape **function** 4-63, 4-110  
 GXDrawBitmap **function** 5-77, 5-79  
 GXDrawCurve **function** 2-41 to 2-42, 2-159  
 GXDrawLine **function** 2-36 to 2-38, 2-158  
 GXDrawPaths **function** 2-57 to 2-58, 2-162  
 GXDrawPicture **function** 6-27 to 6-29, 6-67  
 GXDrawPoint **function** 2-30, 2-158  
 GXDrawPolygons **function** 2-47, 2-161  
 GXDrawRectangle **function** 2-43, 2-160

## GXE, GXF

---

GXExcludeShape **function** 4-64, 4-114

## GXG

---

GXGetBitmap **function** 5-68  
 GXGetBitmapParts **function** 5-53 to 5-54, 5-74  
 GXGetCurve **function** 2-125  
 GXGetLine **function** 2-123  
 GXGetPathParts **function** 2-91, 2-148  
 GXGetPaths **function** 2-132  
 GXGetPicture **function** 6-31 to 6-32, 6-59  
 GXGetPictureParts **function** 6-63  
 GXGetPoint **function** 2-109, 2-119, 2-121  
 GXGetPolygonParts **function** 2-82 to 2-85, 2-144  
 GXGetPolygons **function** 2-130  
 GXGetRectangle **function** 2-127  
 GXGetShapeArea **function** 4-45, 4-88  
 GXGetShapeBounds **function** 4-43, 4-90  
 GXGetShapeCap **function** 3-57, 3-126  
 GXGetShapeCenter **function** 4-43, 4-87  
 GXGetShapeCurveError **function** 3-117  
 GXGetShapeDash **function** 3-66 to 3-70, 3-138  
 GXGetShapeDashPositions **function** 3-81 to 3-85, 3-140  
 GXGetShapeDirection **function** 4-23 to 4-27, 4-68  
 GXGetShapeIndex **function** 2-139  
 GXGetShapeJoin **function** 3-132  
 GXGetShapeLength **function** 4-42, 4-83

GXGetShapeParts **function** 2-152  
 GXGetShapePattern **function** 3-145  
 GXGetShapePatternPositions **function** 3-88 to 3-91, 3-147  
 GXGetShapePen **function** 3-121  
 GXGetShapePixel **function** 5-71  
 GXGetShapePoints **function** 2-140  
 GXGetStyleCap **function** 3-124  
 GXGetStyleCurveError **function** 3-115  
 GXGetStyleDash **function** 3-135  
 GXGetStyleJoin **function** 3-129  
 GXGetStylePattern **function** 3-142  
 GXGetStylePen **function** 3-119

## GXH

---

GXHitTestPicture **function** 6-46 to 6-51, 6-69

## GXI

---

GXInsetShape **function** 4-50 to 4-52, 4-94  
 GXIntersectRectangle **function** 4-105  
 GXIntersectShape **function** 4-61, 4-107  
 GXInvertShape **function** 4-65, 4-116

## GXJ–GXM

---

gxJoinAttributes **enumeration** 3-102  
 gxJoinRecord **structure** 3-101  
 gxLine **structure** 2-105  
 gxLongRectangle **structure** 5-64

## GXN, GXO

---

GXNewBitmap **function** 5-15 to 5-28, 5-66  
 GXNewCurve **function** 2-41 to 2-42, 2-113  
 GXNewLine **function** 2-38, 2-112  
 GXNewPaths **function** 2-58, 2-117  
 GXNewPicture **function** 6-27 to 6-30, 6-57  
 GXNewPoint **function** 2-31, 2-111  
 GXNewPolygons **function** 2-48, 2-116  
 GXNewRectangle **function** 2-43 to 2-45, 2-114

## GXP, GXQ

---

gxPath **structure** 2-107 to 2-108

gxPatternAttributes enumeration 3-107  
 gxPatternRecord structure 3-106  
 gxPoint structure 2-104  
 gxPolygons structure 2-107  
 gxPolygon structure 2-106  
 GXPrimitiveShape function 4-38 to 4-40, 4-79

## GXR

---

gxRectangle structure 2-106  
 GXReduceShape function 4-30 to 4-32, 4-74  
 GXReverseDifferenceShape function 4-64, 4-112  
 GXReverseShape function 4-23 to 4-27, 4-70

## GXS

---

GXSetBitmap function 5-69  
 GXSetBitmapParts function 5-53 to 5-54, 5-75  
 GXSetCurve function 2-79, 2-126  
 GXSetLine function 2-38 to 2-40, 2-79, 2-124  
 GXSetPathParts function 2-91 to 2-93, 2-149  
 GXSetPaths function 2-79 to 2-81, 2-133  
 GXSetPicture function 6-31 to 6-32, 6-61  
 GXSetPictureParts function 6-32 to 6-37, 6-65  
 GXSetPoint function 2-33 to 2-35, 2-79, 2-122  
 GXSetPolygonParts function 2-82 to 2-90, 2-145  
 GXSetPolygons function 2-79, 2-131  
 GXSetRectangle function 2-79, 2-129  
 GXSetShapeBounds function 4-47 to 4-49, 4-92  
 GXSetShapeCap function 3-57 to 3-61, 3-128  
 GXSetShapeCurveError function 3-50, 3-118  
 GXSetShapeDash function 3-66 to 3-70, 3-139  
 GXSetShapeJoin function 3-61 to 3-66, 3-133  
 GXSetShapeParts function 2-93 to 2-99, 2-154  
 GXSetShapePattern function 3-86 to 3-88, 3-146  
 GXSetShapePen function 3-52, 3-122  
 GXSetShapePixel function 5-26 to 5-28, 5-72  
 GXSetShapePoints function 2-142  
 GXSetShapeStyleAttributes function 3-113  
 GXSetStyleAttributes function 3-110  
 GXSetStyleCap function 3-125  
 GXSetStyleCurveError function 3-116  
 GXSetStyleDash function 3-136  
 GXSetStyleJoin function 3-130  
 GXSetStylePattern function 3-144  
 GXSetStylePen function 3-120  
 GXShapeLengthToPoint function 4-42, 4-43, 4-85  
 GXSimplifyShape function 4-33 to 4-37, 4-76  
 gxStyleAttributes enumeration 3-98

## GXT

---

GXTouchesBoundsShape function 4-53 to 4-57, 4-97  
 GXTouchesRectanglePoint function 4-53 to 4-57, 4-96  
 GXTouchesShape function 4-53 to 4-57, 4-98

## GXU–GXZ

---

GXUnionRectangle function 4-106  
 GXUnionShape function 4-62, 4-109

## H

---

hairline dashes 3-78  
 hairlines 3-16 to 3-17  
 halftoning  
   bitmaps 5-30 to 5-32  
 hit-testing  
   picture shapes 6-46 to 6-51, 6-69  
 hollow frame fill. *See* closed-frame shape fill 2-5

## I

---

inclusion. *See* containment 4-3  
 ink objects  
   of bitmap shapes 5-8 to 5-9  
   overriding 6-8 to 6-15, 6-38 to 6-40  
 inseting dashes 3-73 to 3-74  
 inseting shapes 4-50 to 4-52, 4-94  
 inside-frame style attribute  
   as style attribute flag 3-99  
   defined 3-18  
   effect on dash placement 3-73  
   effect on shape with crossed contours 3-53 to 3-54  
 intersection (touching), testing shapes for 4-18 to 4-21,  
   4-53 to 4-57, 4-95 to 4-99  
 intersection operation 4-21, 4-61, 4-105, 4-107  
 inverse shape fills 2-15  
 inversion operation 4-21, 4-65, 4-116

## J, K

---

join attributes 3-26, 3-63 to 3-65, 3-102 to 3-103  
 joins 3-25 to 3-27. *See also* join style property  
   adding to a shape 3-61 to 3-66  
   curve 3-26, 3-103  
   effect of shape fill 3-25, 3-101

- functions for 3-129 to 3-134
- interactions with caps, dashes, patterns 3-33 to 3-35, 3-91 to 3-95
- level 3-26, 3-63, 3-103
- miter 3-102
- miter of 3-27, 3-102
- sharp 3-26, 3-64 to 3-65, 3-103
- standard 3-26, 3-64 to 3-66, 3-102 to 3-103
- join structure 3-101 to 3-102
- join style property
  - defined 3-25
  - functions for 3-129 to 3-134

## L

---

- length of a contour 4-42, 4-83
- level caps 3-24, 3-101
- level dashes 3-105
- level joins 3-26, 3-63, 3-103
- level of picture item 6-19, 6-51
- line geometries 2-9
  - defined 2-17
  - determining 2-123
  - editing 2-38 to 2-40, 2-79 to 2-81, 2-124
  - structure of 2-105
- line shapes
  - converting other shapes to 2-65 to 2-70
  - creating and drawing 2-36 to 2-40, 2-112, 2-158
  - default 2-17
  - defined 2-17

## M

---

- map-transform shape attribute 4-49
  - effect on bitmaps 5-11, 5-40
- miter of joins 3-27, 3-65 to 3-66, 3-102
- multiple references in picture shapes 6-10 to 6-15, 6-40 to 6-44

## N

---

- no-fill shape fill 2-13

## O

---

- objects. *See* ink objects; shape objects; style objects; transform objects
- off-curve control points. *See* control points
- offscreen bitmaps 5-45 to 5-52
- open-frame shape fill 2-13, 2-23
- outsetting shapes 4-52, 4-94
- outside-frame style attribute 3-18 to 3-19, 3-54 to 3-56, 3-99
- overlapping contours
  - creating 2-52
  - effect of shape fill on 2-14 to 2-15, 2-26, 2-63 to 2-65
  - removing 4-33, 4-76
- overriding inks 6-8 to 6-15, 6-38 to 6-40
- overriding styles 6-8 to 6-15, 6-38 to 6-40
- overriding transforms 6-8 to 6-15, 6-38 to 6-40

## P

---

- path contours. *See also* contours; path shapes
  - defined 2-25
  - structure of 2-107
- path geometries
  - control bits 2-25, 2-56
  - defined 2-25
  - determining 2-132
  - editing 2-79 to 2-81, 2-91 to 2-93, 2-133, 2-149
  - with multiple contours 2-60, 2-65
  - with only off-curve control points 2-59, 2-60
  - structure of 2-107
- path shapes
  - approximating with polygon shapes 3-45 to 3-48
  - converting other shapes to 2-74 to 2-79
  - converting to polygon shapes 3-45 to 3-48
  - creating and drawing 2-55 to 2-65
  - default 2-26
  - defined 2-25
  - effect of shape fill 2-26, 2-63 to 2-65
  - with a single contour 2-57 to 2-59
  - with multiple contours 2-60 to 2-65
- pattern attributes 3-32, 3-107 to 3-108
  - as field of pattern structure 3-106
- pattern grid 3-32, 3-107

- patterns 3-31 to 3-33
  - adding to a shape 3-86 to 3-88
  - aligning 3-107
  - definition of pattern structure 3-106 to 3-107
  - effect of shape fill 3-31, 3-106
  - functions for 3-142 to 3-148
  - grid 3-32, 3-107
  - interactions with caps, dashes, joins 3-33 to 3-35, 3-91 to 3-95
  - mapping 3-108
  - positions, determining 3-88 to 3-91, 3-147
- pattern structure 3-106 to 3-107
- pattern style property
  - defined 3-31
  - functions for 3-142 to 3-148
- pen
  - placement of 3-18 to 3-20, 3-53 to 3-56
  - width of. *See* pen width style property
- pen, geometric
  - introduced 3-15
- pen width style property 3-15 to 3-17, 3-51 to 3-53
  - functions for 3-119 to 3-123
- phased dashes 3-69
- picture geometries
  - editing 6-31 to 6-32, 6-63 to 6-67
  - properties of 6-4
  - replacing 6-31 to 6-32, 6-59 to 6-63
- picture hierarchies 6-18 to 6-19, 6-44 to 6-45
- picture items 6-24
  - adding 6-32 to 6-35
  - defined 6-5
  - depth of 6-51
  - level of 6-51
  - multiple references to 6-10 to 6-15, 6-40 to 6-44
  - removing 6-35 to 6-37
  - replacing 6-35 to 6-37
- picture shapes 6-3 to 6-71
  - creating and drawing 6-27 to 6-30, 6-57, 6-67
  - functions for
    - functions specific to picture shapes 6-57 to 6-71
    - other applicable functions 6-52 to 6-56
  - hit-testing 6-46 to 6-51, 6-69
  - introduced 1-20 to 1-22
- pixel image 5-5 to 5-7
- pixel size 5-5 to 5-7
- pixel values 5-5
- point 2-9
- point geometries
  - defined 2-16
  - editing 2-34 to 2-35, 2-79 to 2-81, 2-122
  - structure of 2-104
- point shapes
  - converting other shapes to 2-65 to 2-70
  - creating and drawing 2-29 to 2-36, 2-111, 2-158

- default 2-16
  - defined 2-16
  - disposing of 2-36
- polygon contours
  - defined 2-22
  - structure of 2-106
- polygon geometries
  - defined 2-22
  - determining 2-130
  - editing 2-79 to 2-81, 2-82 to 2-90, 2-131
  - structure of 2-106
- polygon shapes
  - converting other shapes to 2-74 to 2-79
  - creating and drawing 2-45 to 2-55
  - default 2-24
  - defined 2-22
  - effect of shape fill 2-24, 2-51 to 2-55
  - with a single contour 2-46 to 2-48
  - with crossed contours 2-24, 2-50 to 2-55
  - with multiple contours 2-23, 2-49 to 2-50
- port-align pattern attribute 3-33, 3-108
- port-map pattern attribute 3-33, 3-108
- primitive form of shapes 4-12
  - effect of converting to 3-8 to 3-11
  - how to convert to 4-38 to 4-40, 4-79
- primitive shapes 4-9 to 4-16

## Q

---

quadratic Bézier curves. *See* Bezier curves

## R

---

- rectangle geometries
  - defined 2-20
  - determining 2-127
  - editing 2-79 to 2-81, 2-129
  - structure of 2-106
- rectangle shapes
  - converting other shapes to 2-65 to 2-70
  - creating and drawing 2-43 to 2-45, 2-114, 2-160
  - default 2-21
  - defined 2-20
  - effect of shape fills 2-44 to 2-45
- reducing shapes 4-9 to 4-16, 4-30 to 4-32, 4-74
- reverse difference operation 4-21, 4-64, 4-112
- reversing contour direction 4-23 to 4-27, 4-70
- round caps 3-24, 3-59 to 3-61

## S

---

scaling  
   dashes 3-68, 3-104  
   shapes in general 4-47 to 4-49, 4-92

shape attributes  
   map-transform shape attribute 4-49

shape fills 2-12 to 2-15  
   defined 2-12  
   effect of contour direction 2-53 to 2-55, 2-62 to 2-65, 4-23 to 4-27  
   effect on path shapes 2-14, 2-26, 2-63 to 2-65  
   effect on polygon shapes 2-24, 2-51 to 2-55

shape length to point 4-42, 4-85

shapes  
   converting to primitive form 4-38 to 4-40, 4-79  
   insetting 4-50 to 4-52, 4-94  
   outsetting 4-52  
   reducing 4-9 to 4-11, 4-30 to 4-32, 4-74  
   simplifying 4-9 to 4-11, 4-33 to 4-37, 4-76  
   testing for containment 4-18 to 4-21, 4-58 to 4-59, 4-100 to 4-104  
   testing for inclusion 4-58 to 4-59, 4-100 to 4-104  
   testing for touching 4-18 to 4-21, 4-53 to 4-57, 4-95 to 4-99

sharp join attribute 3-103

sharp joins 3-26, 3-64 to 3-66, 3-103

simplifying shapes 4-9 to 4-16, 4-33 to 4-37, 4-76

solid shape fills 2-12. *See also* even-odd shape fill;  
   winding shape fill

source-grid style attribute 3-21, 3-40 to 3-42, 3-98

square caps 3-24, 3-59 to 3-61

standard caps 3-24, 3-59 to 3-61

standard joins 3-26, 3-64 to 3-66, 3-102 to 3-103

start caps 3-57 to 3-61, 3-100

style attributes 3-109 to 3-114  
   auto-inset style attribute 3-20, 3-99  
   center-frame style attribute 3-18, 3-98  
   constants for, defined 3-98 to 3-99  
   device-grid style attribute 3-42 to 3-45, 3-99  
   outside-frame style attribute 3-99  
   source-grid style attribute 3-40 to 3-42, 3-98

style object properties  
   attributes. *See* style attributes  
   cap 3-23, 3-25, 3-57 to 3-59, 3-123 to 3-129  
   dash 3-27, 3-66 to 3-70, 3-134 to 3-141  
   defined 3-5  
   join 3-25, 3-61 to 3-64  
   pattern 3-31  
   pattern property 3-31, 3-86 to 3-88  
   pen width 3-15 to 3-16, 3-51 to 3-53, 3-119 to 3-123

style objects 3-5 to 3-148  
   attributes. *See* style attributes  
   changing directly 3-36 to 3-38  
   changing through shape objects 3-38 to 3-40

curve error. *See* curve error style property

default 3-12 to 3-13

defined 3-5

incorporating into shape objects 4-38 to 4-40, 4-79

of bitmap shapes 5-8

overriding 6-8 to 6-15, 6-38 to 6-40

relationship to shape objects 3-6 to 3-7

style attributes property 3-98 to 3-99

style object properties  
   join 3-25, 3-129 to 3-131

style properties of geometric shapes 3-11 to 3-12, 3-97 to 3-98  
   join style property 3-129 to 3-134  
   pattern style property 3-142 to 3-148

## T

---

text, using as dashes 3-80 to 3-81

touching  
   testing shapes for 4-18 to 4-21, 4-53 to 4-57, 4-95 to 4-99

transfer modes  
   effect on bitmap shapes 5-9, 5-32 to 5-34

transform concatenation 6-19 to 6-23, 6-44 to 6-45

transform objects  
   concatenating 6-19 to 6-23, 6-44 to 6-45  
   in picture shapes 6-38 to 6-40  
   of bitmap shapes 5-10 to 5-11  
   overriding 6-8 to 6-15, 6-38 to 6-40

true inside of a contour 4-9

type conversion  
   defined 2-66  
   to curve shapes 2-71 to 2-74  
   to line shapes 2-65 to 2-70  
   to path shapes 2-74 to 2-79  
   to point shapes 2-65 to 2-70  
   to polygon shapes 2-74 to 2-79  
   to rectangle shapes 2-65 to 2-70  
   table summarizing 2-101, 2-102

## U

---

union operation 4-21, 4-62, 4-106, 4-109

unique items attribute 6-15 to 6-17, 6-43 to 6-44

## V

---

view devices, and bitmap shapes 5-12 to 5-14, 5-45 to 5-51



W, X, Y, Z

---

- winding-number rule for filling shapes 2-14
- winding shape fill
  - compared to even-odd shape fill 2-13, 2-24
  - and concentric contours 2-54
  - defined 2-14
  - and overlapping contours 2-65
- wrapping text to a contour 3-80 to 3-81

---

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final page negatives were output directly from text files on an Optrotech SPrint 220 imagesetter. Line art was created using Adobe<sup>™</sup> Illustrator and Adobe Photoshop. PostScript<sup>™</sup>, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino<sup>®</sup> and display type is Helvetica<sup>®</sup>. Bullets are ITC Zapf Dingbats<sup>®</sup>. Some elements, such as program listings, are set in Apple Courier.

WRITER

Marq T. Laube

DEVELOPMENTAL EDITORS

Laurel Rezeau, George Truett

ILLUSTRATORS

Ruth Anderson, Sandee Karr,  
Mai-Ly Pham

PRODUCTION EDITORS

Pat Christenson, Alan Morgenegg

PROJECT MANAGER

Trish Eastman

LEAD WRITER

David Bice

LEAD EDITOR

Laurel Rezeau

ART DIRECTOR/COVER DESIGNER

Barbara Smyth

Special thanks to Pete Alexander,  
Cary Clark, Dave Good, Josh Horwich,  
Rob Johnson, David Surovell, Chris Yerga

Acknowledgments to Sarah Chester,  
Gary Hillerson, Gary McCue,  
Diane Patterson, Rich Pettijohn,  
Laine Rapin